

Florida Institute of Technology

Scholarship Repository @ Florida Tech

Theses and Dissertations

12-2022

Modeling, Verification, and Simulation of a UAV Swarm Consensus Protocol

Rohit Martin Menghani

Follow this and additional works at: <https://repository.fit.edu/etd>



Part of the [Computer Engineering Commons](#)

Modeling, Verification, and Simulation of a UAV Swarm Consensus Protocol

by

Rohit Martin Menghani

Bachelor of Science
Aerospace Engineering
Florida Institute of Technology
2020

A thesis
submitted to the College of Engineering and Science
at Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Engineering

Melbourne, Florida
December, 2022

© Copyright 2022 Rohit Martin Menghani
All Rights Reserved

The author grants permission to make single copies.

We the undersigned committee
hereby approve the attached thesis

Modeling, Verification, and Simulation of a UAV Swarm Consensus Protocol by

Rohit Martin Menghani

Siddhartha Bhattacharyya, Ph.D.
Associate Professor
Computer Engineering and Sciences
Committee Chair

Juan C. Avendano, Ph.D.
Graduate Faculty
Computer Engineering and Sciences
Outside Committee Member

Brian Lail, Ph.D.
Professor
Computer Engineering and Sciences
Committee Member

Philip J. Bernhard, Ph.D.
Associate Professor and Department Head
Computer Engineering and Sciences

Abstract

Title:

Modeling, Verification, and Simulation of a UAV Swarm Consensus Protocol

Author:

Rohit Martin Menghani

Major Advisor:

Siddhartha Bhattacharyya, Ph.D.

Unmanned Aerial Vehicles (UAVs), particularly electrically powered multi-rotors, are becoming increasingly popular in the entertainment, transportation, logistics, and military sectors. One of the main drawbacks presented by these vehicles at the time of writing is the limited range achieved as a consequence of the limits of battery technology. One common method used to overcome such limitations, is the use of multiple vehicles in cooperation to achieve a certain goal. This application of UAVs is called swarming, where multiple agents can coordinate their actions to fly in a certain formation, to access a certain challenging area, or to fly further. As with any multi-component system, the complexity of swarms equates to multiple points of failure. The risk posed by any uncertainties presented in the highly complex real time mechanics of coordinated flight are unacceptable in safety critical industries. The consequences of critical failures could be significant loss of life, property damage, or environmental harm. Assuring the correct behaviour of UAV swarms is a primordial challenge to be addressed for the

technology to expand its use cases. One of the most robust techniques to verify the correctness of systems is model checking, which allows to verify the behaviour of a system through the systematic inspection of all possible states. The system is verified based on a high-level model representation which allows for the abstraction of implementation details. This approach to software development is not commonly used due to the time and resources investment it requires. The work presented here proposes a framework to model, verify, and provide evidence of correctness for a distributed system of agents with a common objective. Specifically, the framework demonstrates a partially automated process to take a system modeled in the UPPAAL model checker and implement it in the Robot Operating System (ROS) environment. The target system is a swarm of UAVs that travel to a specified goal location while implementing a leader-follower hierarchical structure through a consensus protocol. This document provides an algorithmic representation for the mapping of UPPAAL structures to ROS constructs. The work concludes with the simulation of the ROS implementation of the UAV mission in the Gazebo 3-D robotics simulator.

Table of Contents

Abstract	iii
List of Figures	viii
List of Tables	xi
Acknowledgments	xii
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	6
1.3 UPPAAL and ROS	7
2 Related Works	9
3 Research Methodology	25
3.1 Problem Statement Elaboration	25
3.2 Framework	26
3.3 Formal Model	27
3.4 Assumptions	30
3.5 Translation	31
4 Technical Specifications	32

4.1	Matlab Drone Dynamics	32
4.1.1	Inputs and Outputs	33
4.1.2	Matlab Script Modifications	36
4.2	UPPAAL Description	37
4.3	UPPAAL Global Declarations	39
4.4	UPPAAL Template Descriptions	41
4.5	Election Protocol	45
4.6	Properties Verified	49
5	ROS Implementation	54
5.1	ROS Concepts	54
5.2	Automated Translation	56
5.3	Manual Translation	59
6	Validation	63
6.1	Simulation Setup	63
6.2	UPPAAL Symbolic Simulator	66
6.3	Simulation Test Cases	67
7	Conclusion and Future Work	78
7.1	Conclusion	78
7.2	Limitations	80
7.3	Future Work	82
	References	83
A	Drone Dynamics Matlab Script	89
B	Uppaal Drone Swarm System XML	111

C	ROS Base Classes	142
D	ROS Node Control Files	163

List of Figures

1.1	IoT Interaction Patterns [9]	2
1.2	Amazon’s Latest Drone Design MK27-2 [38]	3
1.3	Drone Swarm Example [46]	3
1.4	Model Checking Flow Chart	6
2.1	System Design Flow [26]	10
2.2	Code Generator Overview [26]	11
2.3	Publisher-Subscriber System Timed Automata Model [14]	12
2.4	UPPAAL Implementation of Publisher-Subscriber Timed Automata [14]	13
2.5	Drone Template [34]	15
2.6	Global Variable Translation [34]	16
2.7	Server Template Translation into C++ Class [34]	18
2.8	Swarm UPPAAL Model [30]	19
2.9	Swarm Probabilistic State Machine [22]	22
2.10	LandShark Robot Control System Architecture [28]	23
2.11	Code Generation Process and Tools [28]	24
3.1	Framework Diagram	27
3.2	Initial Mission Scenario	28
3.3	Drone Trajectory	30
4.1	Sample Drone Trajectory Optimizing Time [16]	34

4.2	Script Added to Matlab Code-base	36
4.3	Time of flight for different trajectories	36
4.4	UPPAAL Project Structure	37
4.5	Edge Annotations	38
4.6	UPPAAL Drone Template	41
4.7	UPPAAL MissionControl Template	43
4.8	UPPAAL System Declarations	45
4.9	Member Election Function	46
4.10	Member Election Results Processing	47
4.11	Checking for Consensus	47
4.12	Leader Election Function	48
4.13	Leader Election Results Processing	49
4.14	UPPAAL Satisfied Properties	50
5.1	ROS Communication Architecture [25]	56
5.2	Automatically Generated ROS Parameter Server YAML File [33]	58
5.3	Base Class Voting Function Example	59
6.1	ArduPilot SITL Architecture [3]	64
6.2	SITL Simulation Setup [45]	65
6.3	UPPAAL Symbolic Simulator Snapshot	66
6.4	UPPAAL Global Parameters	68
6.5	ROS Parameter List	68
6.6	Drone 0 and Drone 2 Initially In Swarm State	69
6.7	Drone 3 Initially In Swarm State	70
6.8	ROS Drones In Swarm	70
6.9	First Leader Election Votes	71

6.10 Drone 0 Elected Leader	71
6.11 Drone 0 Leader in ROS	72
6.12 Drone 7 Elected As New Member	72
6.13 Drone 0 Idle State	73
6.14 Drone 7 Elected Leader	74
6.15 Drone 7 Leader State	74
6.16 Drones at Position 60 in x	75
6.17 Drone 7 Dropped from Swarm and Drone 1 Joins	75
6.18 Drone1 Becomes Leader	76
6.19 Drone 3 Arrives at Goal First	76
6.20 ROS Drones at Goal	77
6.21 Mission Accomplished	77

List of Tables

4.1	Verification Time Metrics	53
5.1	UPPAAL to ROS High-Level Mapping	59
5.2	UPPAAL to ROS Low-Level Mapping	60

Acknowledgements

Firstly, I would like to thank the people who gave me every opportunity I ever had, my parents Rakesh Menghani and Jasmine Menghani. I want to thank my sister Natasha Menghani for being a role model and the reason why I was brave enough to leave home. Thank you Martina Farachio for keeping me sane throughout my stress-filled years at college and your unconditional love.

Thank you Deep Patel for opening doors for me every time I needed help, and for treating me like family alongside Aashna Patel. Thank you Dr. Nicolas Jaramillo and Darshan Yadav for teaching me how to be a decent GSA, and for being like two elder brothers. Thank you Dr. Juan Avendano and the CAMID team for supporting me in my last semester and being ever so patient as I wrapped up my work. Thank you Dr. Siddhartha Bhattacharyya for your guidance on this work and your amazing courses. Thank you Mamoon Syed for the endless knowledge.

Gracias a "Terminemos la ... Tesis" por las noches de trabajo y risas sin las cuales no estaria escribiendo esto en este momento. Gracias a mi familia de LASA por las interminables anecdotas. Gracias a toda mi gente de Argentina que nunca dejo de hacerme el aguante.

Thank you Florida Institute of Technology for five unforgettable years.

Chapter 1

Introduction

1.1 Motivation

Intelligent systems have become embedded in our daily lives to the extent that the average person uses them unknowingly. This is particularly enhanced by the popularity of IoT (Internet of Things) devices, which increasingly use machine intelligence to give systems autonomy [9]. The interactions involving IoT devices between the social, cyber, and physical world can be seen on Figure 1.1. The work herein is focused on autonomous aerial vehicles and the cooperation amongst themselves to achieve target system behaviours. Such systems are relevant to the Urban Air Mobility (UAM) and AAS (Advanced Aerial Systems) fields which encompass novel air transportation and delivery systems. In fact, it is estimated that the United States UAM market will reach \$18.8 billion by 2035 due to the revenue these new systems will produce [31].

Some of the biggest names developing UAV delivery technologies are Amazon, UPS, and Wing. The three of them have already gained approval from the Federal Aviation Administration (FAA) to operate drone delivery fleets[32]. These companies are focusing on the delivery of lightweight packages that are close to their final destination. An

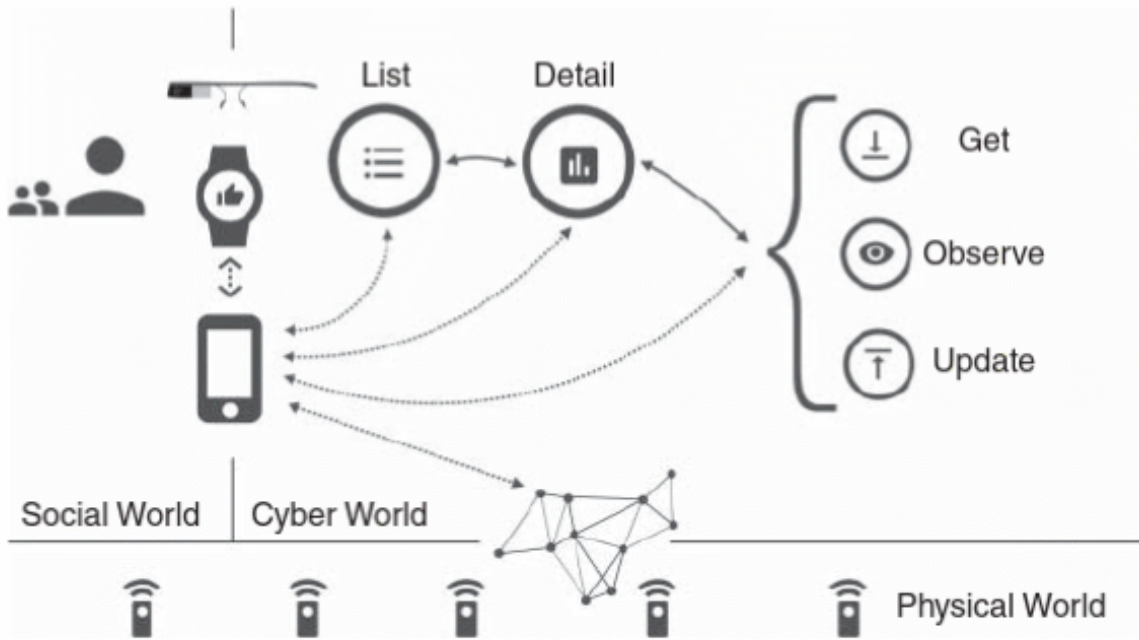


Figure 1.1: IoT Interaction Patterns [9]

example of an Amazon delivery drone is shown in Figure 1.2. The main limitation for such delivery operations is drone battery duration, which is shortened by carrying a payload. The same problem affects larger Electric Vertical Takeoff and Landing (eVTOL) transportation systems currently being developed by large firms like Embraer, Uber, and Joby [5].

One of the technologies developed to overcome the limitation presented by singular electric aerial vehicles is aerial swarms. These consist of multiple aerial vehicles cooperating in a manner that helps them achieve tasks that would be challenging for singular agents. UAV swarms constitute distributed systems in which vehicles gather information about the environment and communicates with other vehicles to coordinate actions that achieve a certain goal [2]. Figure 1.3 shows a drone swarm flying in a rectangular formation. In swarms where aircraft are organized in a leader-follower configuration, distributed consensus is one of the core principles. This concept is typically



Figure 1.2: Amazon's Latest Drone Design MK27-2 [38]

used to establish the agreement between a leader and its followers about new vehicles joining the established swarm or group. The agreement protocol can be described by a plethora of popular distributed consensus algorithms.

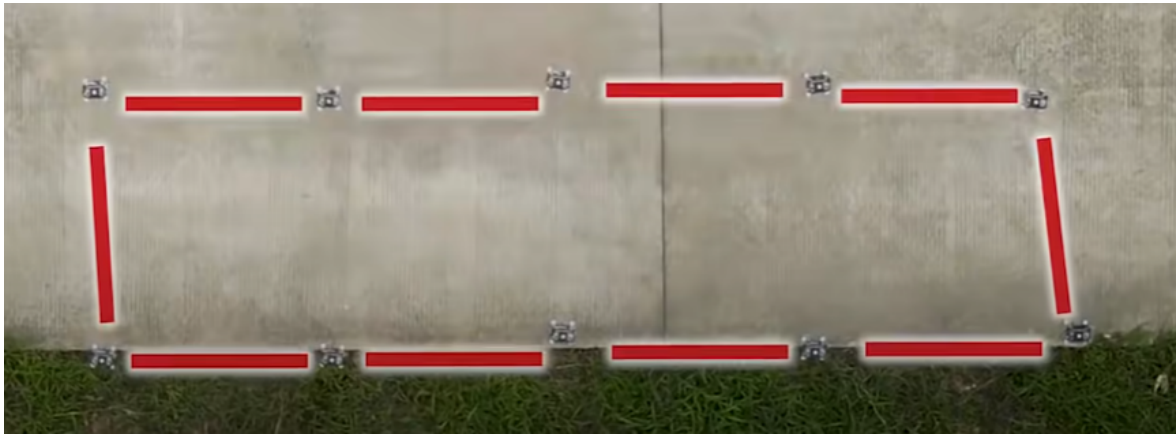


Figure 1.3: Drone Swarm Example [46]

When dealing with complex systems such as swarms, one of the biggest challenges

is to ensure correct behaviour of each of the agents in the network. To achieve this, a relatively rigorous method that includes the theories from the field of formal methods have gained popularity one such method is model checking. It is a formal verification technique that allows for the behavioral properties of a system to be verified based on a model through the systematic inspection of all possible states [6]. The process flow chart for model checking is shown in Figure 1.4. The process of creating a model of a system is abstracted from the actual details of the implementation, which means that a higher-level representation is developed. The benefit of this is that one can rapidly (relative to implementing the whole system) model and check the system before getting entangled in the implementation phase. This is an advantage in itself because bugs and faulty requirements can be found early in the design phase cutting development costs significantly.

The benefits of developing a software system by including formal approaches is highlighted by the findings published by the Standish Group in their CHAOS reports. On their latest report, published in 2020, they found that 65% of IT projects are not successful in terms of budget and timeline. Particularly, they estimate that more than half of the projects evaluated end up costing 189% of their original budget proposals, almost double [12]. The prime reason for this is the introduction and fixing of errors at different stages of the software development cycle. To put this into perspective, according to the Consortium for Information & Software Quality, the cost of finding and fixing software bugs in the US was around 600 billion dollars in the year 2020 [24].

Another problem related to the increased cost of software development is error injection. Most errors in software systems are introduced in the early phases of development, where addressing them is the least costly. However, even though less abundant, error injection in later development stages is just as common. The issue is that fixing errors introduced in the testing or release phases can be hundreds of times more expensive

[18]. Thus, introducing formal approaches in the early stages of software projects can help economically address the great amount of errors injected and also avoid future errors.

The advantages of using formal methods are perfectly suited to address the challenges of vehicle certification in safety critical industries like the UAM industry. Safety critical industries are ones where failure in systems could result in death, injury, property damage, or environmental harm [37]. What is more, UAM missions like air taxis will involve aircraft that combine a range of new technologies. For instance, UAM eVTOL aircraft will likely feature complex Distributed Electric Propulsion (DEP) systems and intelligence-based autonomous flight control systems [20]. This means that the traditional aircraft certification process mandated by the FAA is not applicable. Hence, the incorporation of formal methods to verify and assure aircraft system behaviour will become pivotal in the certification processes. What is more, since the companies designing and developing eVTOL aircraft for UAM applications are the same ones that will be providing services, it is in their best interest to prove the safety of their product. It would be fatal for the industry to have any safety incidents during its inception phase because it would negatively impact the public’s willingness to try their new technology. A number of research efforts have been conducted to apply formal methods to UAM components such as autonomous Air Taxi vehicles [13], flight scheduling [44], and high density trajectory planning [42].

The only drawback associated with formal verification is that it adds an extra layer to the development process that might increase the time of production. Although the extra time invested in verification might pay off when debugging and testing times are cut back, there is no framework that allows for a seamless transition from model checking to implementation. To this extent, the goal of the work presented hereby is to showcase a partially automated framework to write implementation code for a swarm

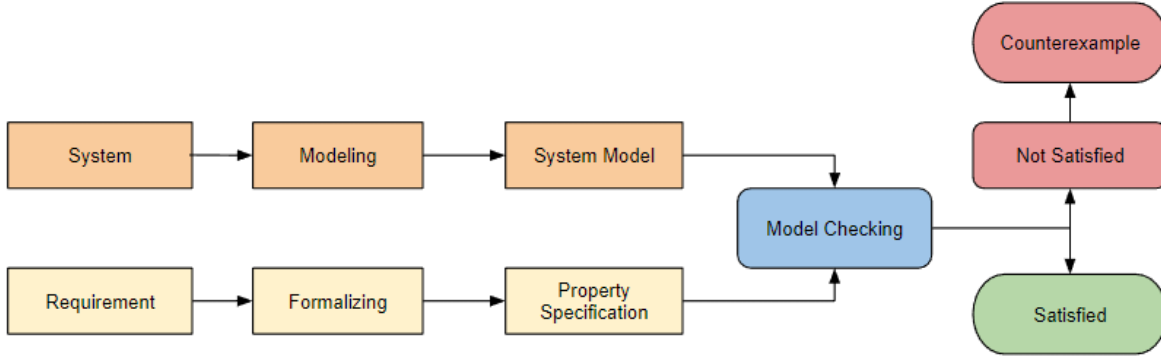


Figure 1.4: Model Checking Flow Chart

based on a verified model. In particular, this work presents an algorithmic mapping of an implementation in the Robot Operating System (ROS) based on an UPPAAL model. The target system that will be used as case study for the framework is the consensus protocol used by a UAV swarm that achieves a leader-follower configuration where new agents join the network.

1.2 Problem Statement

As introduced in the previous section, the main swarm behaviour explored in this work is the distributed consensus scenario. The application of this concept used as basis for modeling and simulation is the agreement between leader and follower aircraft in a swarm taking part of wild fire fighting mission. The swarm engages in two voting scenarios: one for electing new swarm members and one for electing a swarm leader. There are two types of criteria used as basis for an agent joining a drone network: physical parameters and logical parameters. The physical parameters used for a UAV to join the swarm are:

- Within a 10 meter radius
- More than 50 units of battery

- More battery capacity than drone in the swarm that has the same capability
- Fastest drone to arrive at swarm location

On the other hand, the logical conditions considered as swarm joining parameters are the capabilities that each drone have to contribute to the swarm. For example, if the mission goal is to put out a small wild fire, the mission will require a drone with the capability to drop water or fire retardant on the flames. To this extent, five different drone capabilities are defined: surveillance, communications, medical supplies, water dispensing, and chemical dispensing.

The flight scenarios addressed in this work will be gradually increasing in complexity. For example, the first scenarios will only involve three agents where there is a leader and a loyal wing-man connection pre-established. The third agent will then request to join the network and a decision will be made based on the consensus algorithm. More advanced scenarios would involve two different groups of agents where the leaders of each group would have to agree to merge and hand off the leadership. Such problem scenarios are relevant to real world situations where not only the physical interactions between agents need to be considered but also the logical interactions (e.g. capabilities, resources, etc).

1.3 UPPAAL and ROS

As defined on the UPPAAL software website, "UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types" [21]. These networks of timed automata, which represent system behaviors, are described using the UPPAAL description language. The validation of the created models is achieved with the simulator tool, which enables examination of possible dynamic executions of a system. Finally,

verification is attained through the model checker’s exhaustive examination of the dynamic behaviour of the system. This last feature allows for checking of invariant and reachability properties by state space exploration. The use of this software is suitable for systems that can be represented as non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables [21]. UPPAAL is developed by a partnership between the Department of Information technology at Uppsala University, Sweden and the Department of Computer Science at Aalborg University in Denmark.

ROS is an open source software framework for programming robots [19]. It provides a hardware abstraction layer that developers can use to build robotics applications and software tools to visualize robot data [23]. The foundation of the ROS framework is a message passing (synchronous or asynchronous) middleware where processes communicate and exchange data amongst themselves. In addition to this, ROS software is arranged in packages, enabling modularity and re-usability as different robots can access the same packages for capabilities [19]. Some of the quoted advantages for using ROS are its collaborative development community, variety of programming language support (C++, Python, etc), third-party library integration (e.g Open-CV), simulator integration (e.g. Gazebo), code testing framework rostest, scalability, and customizability.

Combining these two frameworks into a single process for robotics system development would provide significant benefits in terms of assurance, cost, and time. This is exactly the goal of this work, showcasing a framework for creating low level implementation code for a ROS system based on a verified UPPAAL formal model.

Chapter 2

Related Works

There are many research efforts that produced works related to the goal of this study. Particularly, [26] proposes an automatic code synthesis method that can generate C++ code for ROS, based on a verified formal model of a robot system. The researchers first modeled the behavior of the robot system as timed automata. Then the relevant properties, and the functional and architecture requirements were manually translated into temporal logic formulas in the UPPAAL query language. The model was then verified to assure correctness. Finally, the code generator developed by the team automatically synthesized the model into executable C++ code for ROS.

The generated code abides by the ROS instructions scheduling mechanism involving node communication rules, function prototype design, and driver interface configuration. The code generator is based on a Java parser which achieves the low level abstraction from model to code by using the DOM4j library. This is an open source library used to parse XML files. The input for the parser is the model XML file which can be exported from UPPAAL. The parser creates static and dynamic look up tables that contain information on the model states and transitions. Another essential component of the synthesizer is the Controller, which implements high-level abstraction of

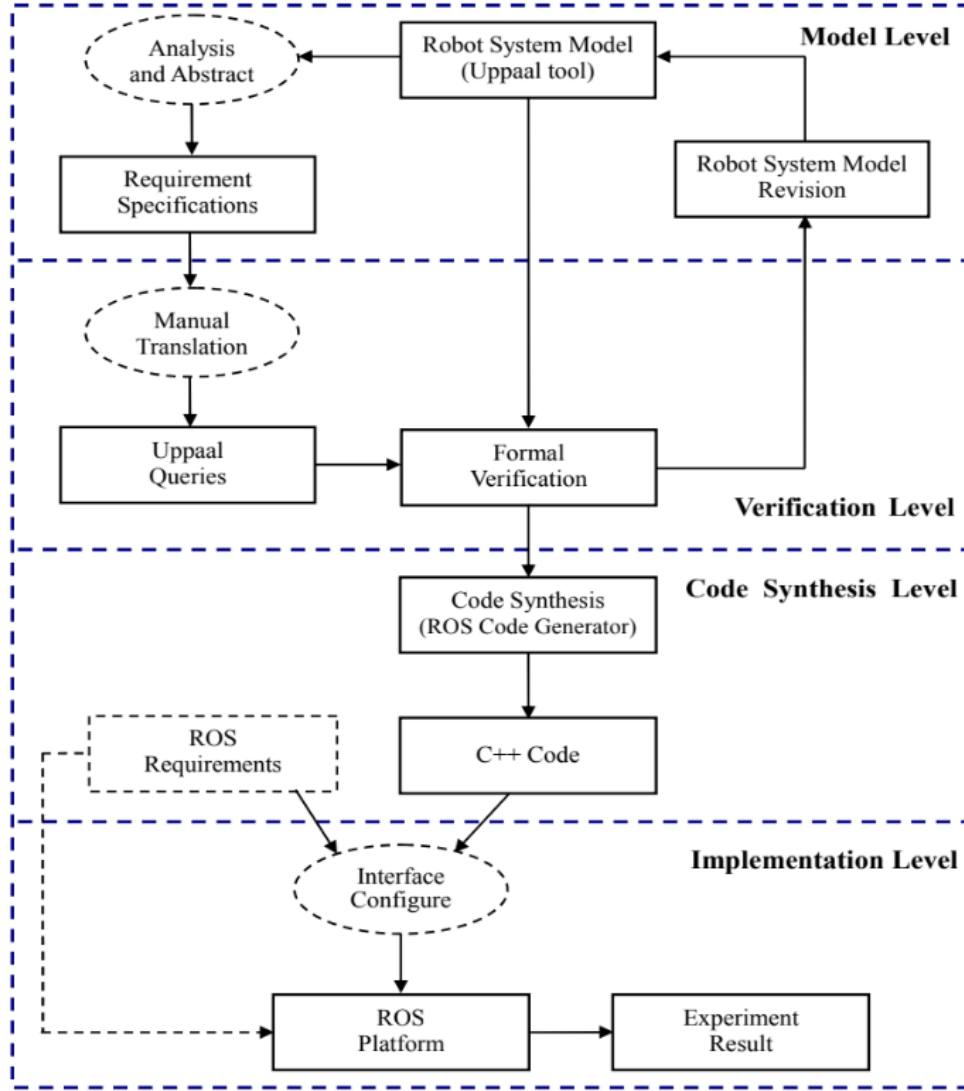


Figure 2.1: System Design Flow [26]

the model. Particularly, it creates an abstract specification of the branch and transition logic of the model. Lastly, the Storer outputs the C++ files containing the generated code. The results of the automatic code synthesizer were tested using a robotic arm with seven degrees of freedom simulated in the Gazebo physics simulator.

The main difference between [26] and this work is that they considered a single robot as their use case. The work presented here involves multiple agents that are

part of a network. What is more, the proposed system involves modeling a consensus algorithm for leader election which adds to the overall complexity.

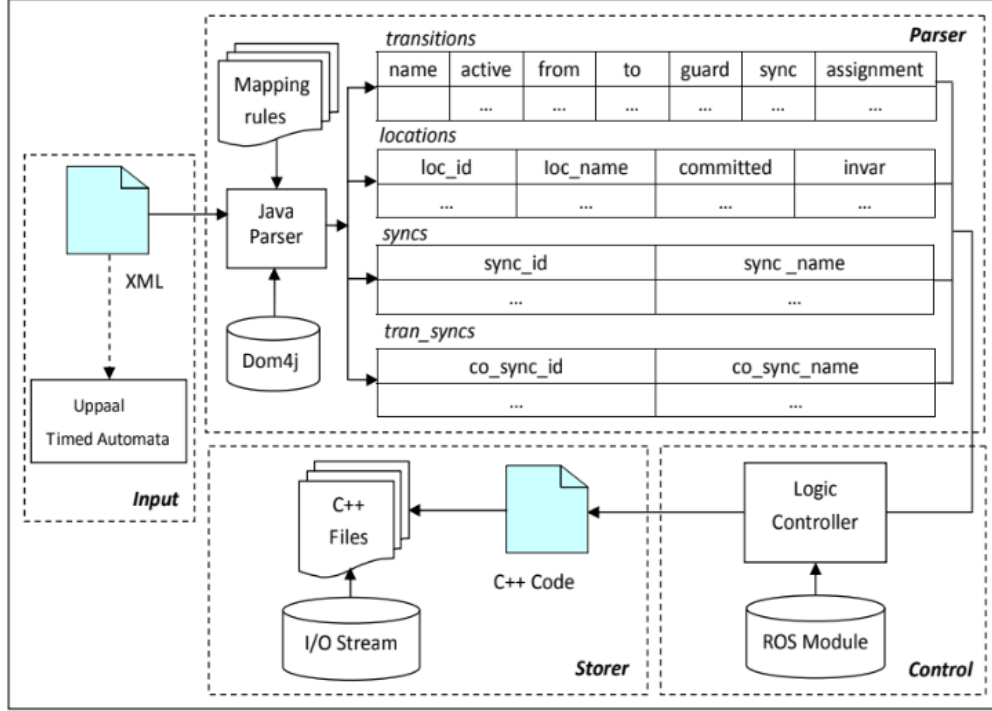


Figure 2.2: Code Generator Overview [26]

Another paper that addresses the ROS implementation of verified UPPAAL models is [14]. In this work, the authors first implement the target ROS system and then verify it using the UPPAAL model checker. The research group proposes an approach to model and verify the code used by the Kobuki robot in ROS. The focus of their work is the communication between nodes, which means low level parameters, such as queue sizes and timeouts, are taken into consideration. Their goal is to identify problematic configuration parameters used in the Kobuki source code.

The authors describe the process of verifying ROS applications using a simple example consisting of publisher-subscriber interactions. The first step they took was to extract key parameters from the target ROS application source code. Specifically, the

authors consider nodes, topics, and messages where key parameters are identified as publishing rates, spin rates, channel message transmission times, and callback processing times. For a better understanding of these concepts please refer to section 5.1. The extraction of these key parameters is not automated which represents a major gap in their work as compared to the work proposed in this Thesis.

The next step outlined in [14] is to model the ROS application as timed automata using the key parameters extracted in the previous step. The model of the simple publisher-subscriber system is shown in Figure 2.3.

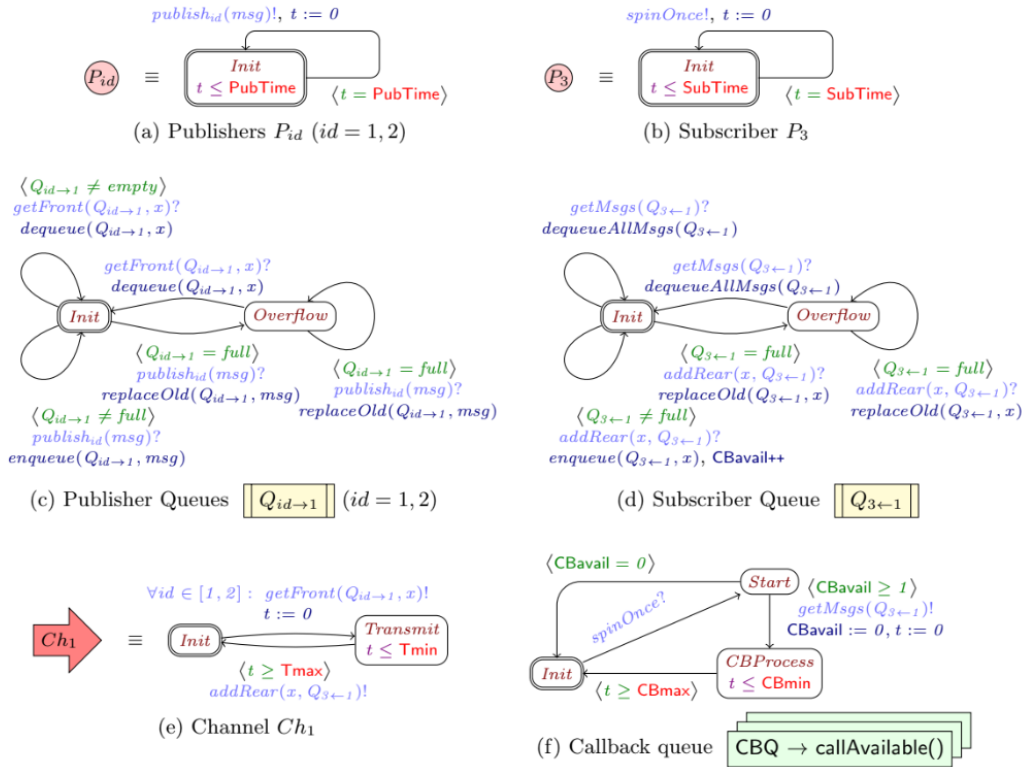


Figure 2.3: Publisher-Subscriber System Timed Automata Model [14]

Finally, based on the formal models, the researchers implemented the simple example of a publisher-subscriber system in UPPAAL. The properties verified in the model checker were mainly checking for queue overflow. The parameters that were varied for

the checking process were queue sizes and time constraints. The UPPAAL templates corresponding to the network of timed automata shown in Figure 2.3, are shown in Figure 2.4.

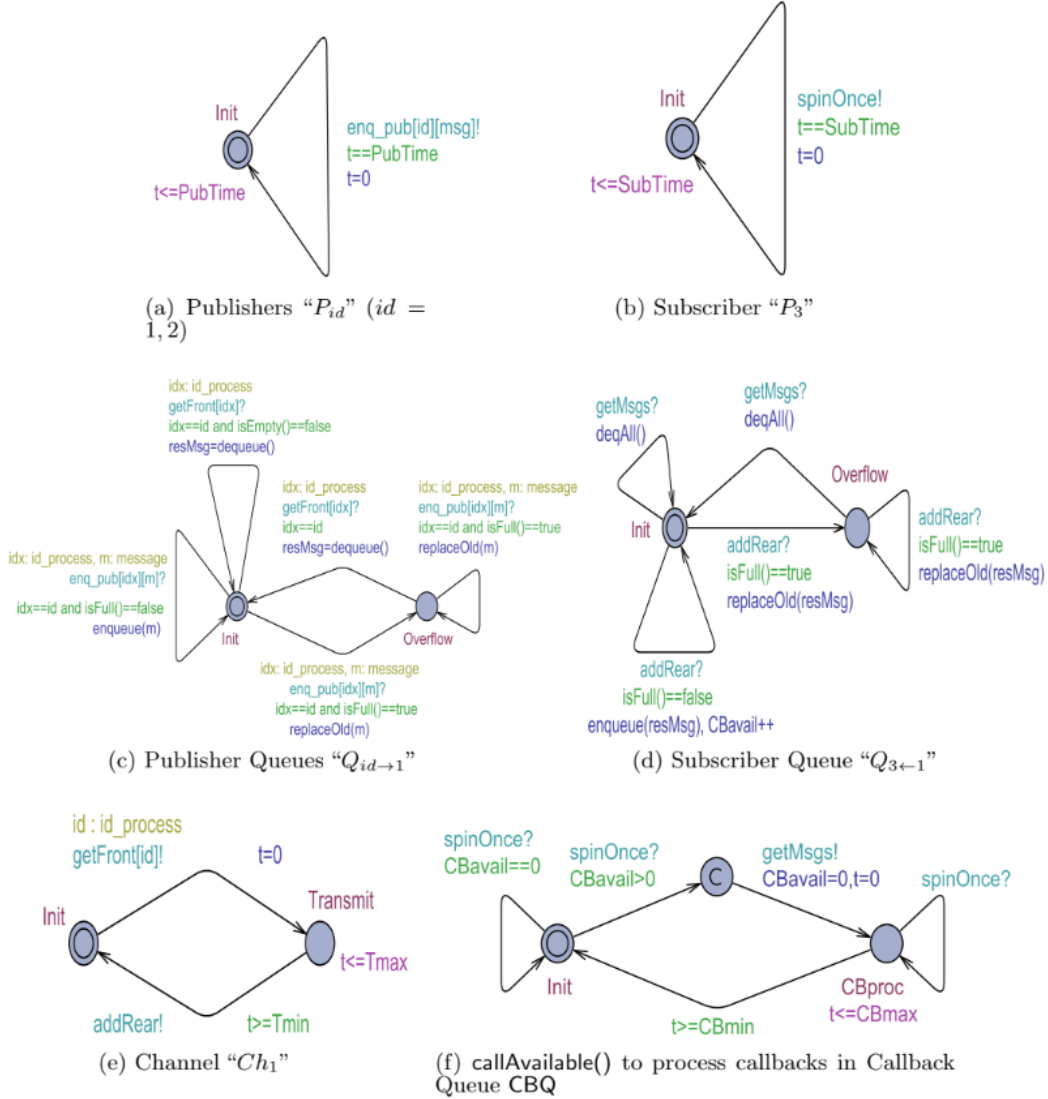


Figure 2.4: UPPAAL Implementation of Publisher-Subscriber Timed Automata [14]

The authors go on to repeat the process explained above using the Kobuki source code as a complex case study. There are various differences between [14] and this work. Firstly, the approach by Halder et al. is in the reverse order of the one proposed

here. The goal of this work is to be able to first model a system in UPPAAL and then translate it into ROS. The benefits of undertaking such a development order are explained in section 1.1. What is more, the framework developed in this work targets automatizing the translation from the formal UPPAAL model to the ROS implementation. Another major difference between these works is that [14] focuses on low level communication aspects of the systems modeled, while this work focuses on higher level robot interactions. In fact, the system proposed in this work consists of a multi agent network in contrast to the single robot case study evaluated by Halder et al.

The research effort described in [34] is also strongly intertwined with the goals of this work. The authors aimed to develop a framework to translate high level abstracted UPPAAL models into the ROS environment. Much like this Thesis, the authors of the paper "seek to accelerate development cycle in transitioning from formally verified systems to simulation" [34]. More pertinently, the scenario used for development is a distributed drone network representing a typically Urban Air Mobility situation. The paper referenced discusses the modeling of the distributed protocol from a formal approach. The modeled distributed consensus protocol involves drones coordinating to agree on which agent will satisfy a certain request established by a server. The leader election is based on the shortest distance from each agent to the location to be serviced.

The UPPAAL model created by the research group consists of four templates: `Sensor_input`, `Drone`, `Server`, and `Input`. The `Input` template randomly generates requests to be serviced by drones and synchronizes with the `Server` template to pass on the request. The `Server` template is used by each drone to authenticate the service request, and also triggers the consensus mechanism for drones to decide who is to serve the request. The `Drone` template models the distributed behavior of the drone network.

Firstly, it conducts initial drone checks to determine if they are ready for the mission. This is when the Sensor_Input template comes into play. Then the Drone template also models the calculation of the shortest distance to the service request and finally determines which drone is closest based on consensus. The Drone template can be seen in Figure 4.6 below.

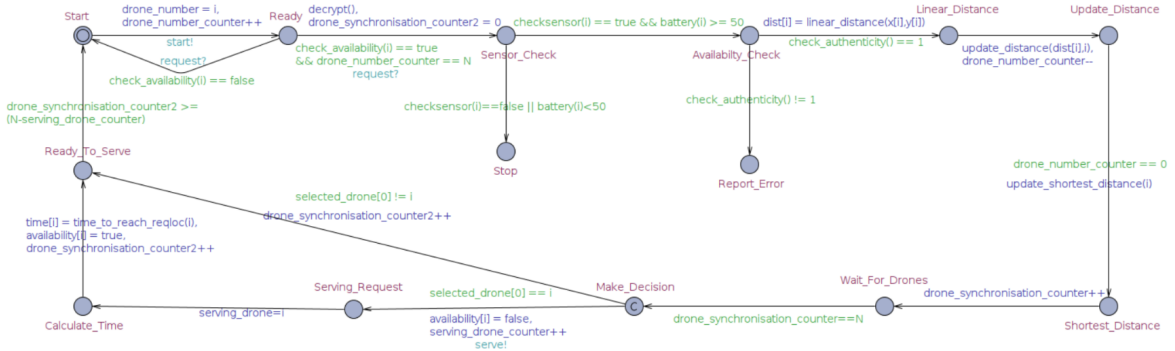


Figure 2.5: Drone Template [34]

Although the research effort concluded on a manual translation from UPPAAL to ROS, the paper thoroughly describes the translation process. The authors first address the translation of global variables used in UPPAAL to ROS. They identified the ROS counterpart to be the parameter server, which stores all the shared variables. The implementation for the translation of global variables involved storing these in a YAML file which was loaded at runtime through path specification in ROSlaunch file. This allows any node within ROS to have access to the global shared variables. Figure 2.6 shows the global variables in UPPAAL (top) and their translation as declared in the YAML file (bottom).

Next, the research work addresses the translation of UPPAAL templates (transition systems) and local declarations. Their approach involved creating two distinct C++ files, one for the local declarations and one for the logic represented by the transitions between nodes. The translation of the UPPAAL local declarations was achieved by

```

// Place global declarations here.

const int N=3;

int altitude[N];
int fuel[N];
int temp[N];
bool technical_sensor[N];
int bat[N];
int drone_number;
int drone_number_counter = 0;
int drone_synchronization_counter = 0;
int drone_synchronization_counter2 = 0;
int serving_drone_counter = 0;

N: 3
altitude: [0,0,0]
fuel: [0,0,0]
temp: [0,0,0]
technical_sensor: [false, false, false]
bat: [0,0,0]
drone_number: 0
drone_number_counter: 0
drone_synchronization_counter: 0
drone_synchronization_counter2: 0
serving_drone_counter: 0
time_server: 0
time: [0,0,0]
global_key: 0

```

Figure 2.6: Global Variable Translation [34]

creating classes with public and private variables or functions. The mapping in this case was straight forward given that the UPPAAL local declarations are formatted in C. According to the authors, the only challenge with this part was choosing the appropriate data structures to go with each C declaration. For example, they chose to represent arrays from the UPPAAL declarations as C++ vectors. If any of the classes representing local template declarations needed the ability to read or write to global variables in the parameter server, they were assigned a private ROS Node Handle. An

example translated class for the Server template can be seen in Figure 2.7.

As the paper explains, the only data type from the UPPAAL declarations that does not directly map to a C data type are channels. The research team successfully modelled UPPAAL synchronization channels in ROS by declaring a global integer which would only have values 0 or 1. The corresponding ROS nodes which need to listen to this "channel" achieve it using the "while(ros::ok())" loop. Executions are triggered when the integer value is set to 1. States in the timed automata model were represented using Boolean variables.

The complimentary file representing each UPPAAL template was used to instantiate the template classes, and also represent the logic defined in the transition diagrams. The guard conditions represented on the edges of the timed automata network were represented using if-else-if conditional statements. When these statements involved global parameters, the authors utilized the node handler methods inherent to ROS, set and get. These files were executed in ROS by spawning them as ROS nodes defined in the launch file. The mapping was taken directly from the System Declarations in the UPPAAL model.

The research team concludes by identifying key limitations in their approach to the problem. The main one is that the translation task is not automated which can result in slow and tedious development. The main goal of this Thesis is to improve on these aspects and achieve a automatic verified framework to go from a verified UPPAAL model to an implemented ROS simulation. Another key limitation identified by the research group is that the correctness of the translation from UPPAAL to ROS was not verified.

Another work related to this Thesis is "On the robustness of consensus-based behaviors for robot swarms" by Majda Moussa and Giovanni Beltrame [30]. The work presented focuses on using statistical model checking to model and assess the robust-

```

class Server
{
private:
    int time_value;
    int local_key;
public:
    Server()
    {
        nh = ros::NodeHandle("-");
        this->time_value = 20;
    }
    void sendcoordinates(int x, int y)
    {
        vector<int> y;
        nh.getParam("/y", y);
        vector<int> x;
        nh.getParam("/x", x);
        int y_coord;
        nh.getParam("/y_coord", y_coord);
        int x_coord;
        nh.getParam("/x_coord", x_coord);
        x_coord = x;
        y_coord = y;
    }
    void encrypted_key(int i)
    {
        int global_key;
        nh.getParam("/global_key", global_key);
        local_key = (i / (i/5))* (i /25);
        local_key = local_key / 3;
        global_key = local_key - 23;
    }
};

```

Figure 2.7: Server Template Translation into C++ Class [34]

ness of consensus based behavior from a communication standpoint. The approach was validated using two swarm scenarios: leader election and allocation of tasks. The authors argue that very few works use formal methods for evaluation of robustness of robotic swarms as critical systems. They claim to bridge the gap by providing methodology for the formal modeling of algorithms which implement consensus. The focus of the work is on robustness which is defined as the degree to which the system keeps its tasks in the presence of partial failure or other abnormal conditions (bugs, crash,

network issues). A novel formal model of generic consensus system, called Virtual Stigmergy (VS), is built to ensure its robustness to packet drop probability and robot failure probability.

The first (simpler) scenario evaluated is that robots must agree on the highest value of a given parameter (electing leader). Then, the complex scenario is based on partitioning of a set of tasks based on a bidding approach. The main assumption behind this work is that robots failing is equivalent to robots going out of communication range. Under this assumption, the abstraction level allows the authors to ignore the exact position of robots for modeling purposes. The research group's approach is using the UPPAAL statistical model checking tool (SMC UPPAAL) and modeling systems as a network of priced timed automata PTA. These are automata extended with clocks, clock rates, probabilistic delays, and weights. Where Clocks measure the delay between different actions in the model, evolve synchronously and continuously, while Clock Rates quantify the clock evolution in the different states of the model.

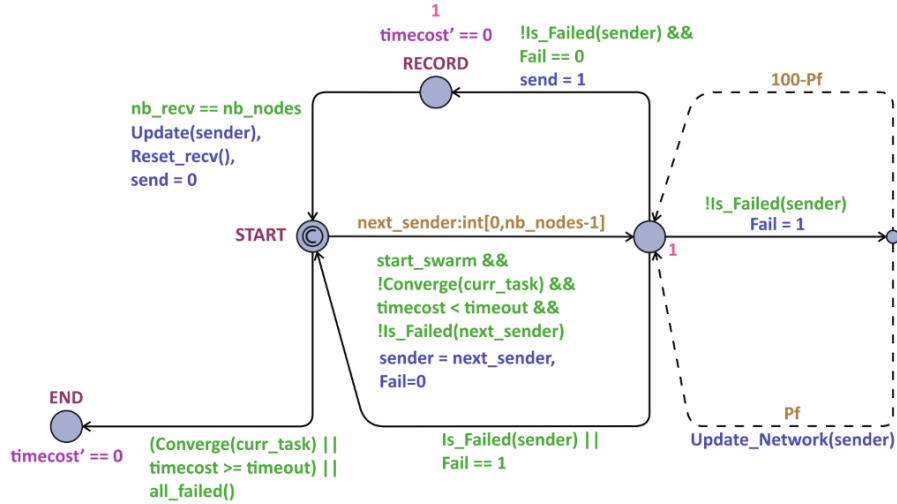


Figure 2.8: Swarm UPPAAL Model [30]

The authors of the paper used two strategies to model consensus problems. The

first strategy was to model an individual agent template where a swarm of n robots is represented by a network of n identical PTA. This first modeling strategy provided insight of agent interaction using broadcasting channels but resulted to be less scalable because of state space explosion. The second strategy was a Swarm based model where a single template represents the swarm network as one automaton. Although this model resulted to be less expressive it reflected better scalability because there was no state space explosion. Both models were validated successfully using the leader election scenario where conflicts in consensus did not arise. Their UPPAAL model representing a Swarm in a single template is shown in Figure 2.8.

The main difference between [30] and this Thesis is that the authors of the paper summarized above relied on Statistical Model Checking as means of probabilistic verification. What is more, their simulations were implemented in the Buzz programming language and using the ARGos physics simulator. It is also important to note that their implementation of the verified model was done manually and not through an automated process.

Another paper surveyed for this project was “Analyzing robot swarm behavior via probabilistic model checking” by Savas Konur, Clare Dixon, and Michael Fisher [22]. This research paper explores the field of probabilistic model checking by an examination of state based control algorithms in of foraging robots. The authors target existing robot swarm algorithms for foraging robots, describe the control algorithm within each robot as a probabilistic model, and then automatically analyze all possible runs through a system of multiple robots using the PRISM model checker. While individual robot properties could have been verified, the focus was placed on the overall swarm behavior. According to the authors, the major contribution was to explore formal verification of probabilistic and population based models of swarm behavior via model checking.

Since robot control algorithms usually require not only the formalization of temporal

behavior (standard model checking), but also uncertain behaviors, the research group used the probabilistic model checker Prism. This model checker supports probabilistic queries as well as quantitative structures defining costs and rewards.

The researchers took two approaches to modeling the system: Microscopic and Macroscopic [22]. In the Microscopic or agent based model, each robot was modeled in detail and the Swarm model was constructed from the product of all the individual robot models. On the other hand, the Macroscopic or population based model, abstracts away from details of individual robot behaviors and models the population as whole.

Specifically, the Microscopic approach consists of instantiating the transition system for each robot in the swarm, and taking the product of all these to provide overall swarm behavior. Global variables are used to calculate the number of robots at each state at any time. Although this is a good approach to observe behavior of single agents, it proved to be inefficient because the product of the state space quickly caused state explosion. What is more, as the researchers concluded, verification of properties became even more computationally intensive, and memory requirements rapidly became unsatisfiable.

On the other hand, the Macroscopic approach consisted of implementing the counting abstraction approach. This is also called population model and it is particularly useful if there are many identical independent processes. It allowed the group to abstract away from low level probabilistic details and just consider global population behavior. Since each robot considered was identical, the system was modeled by one state machine. A counter was added to each of the states to record how many robots were actually in that state at any given moment.

Another useful concept introduced in this paper was swarm energy. The team quantified energy or battery consumption of each robot for each time step. The paper

also explored collision avoidance by having additional states associated with each of the agent states. The transition diagram for their system can be seen in Figure 2.9.

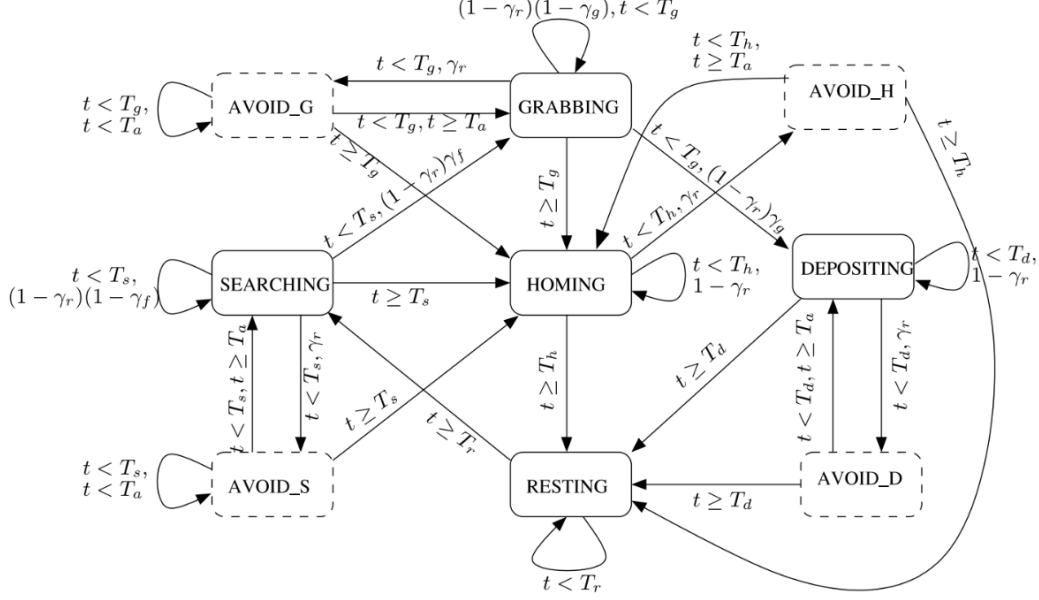


Figure 2.9: Swarm Probabilistic State Machine [22]

The most obvious difference between the paper introduced above and this Thesis is that the research team in [22] did not use the UPPAAL model checker but rather used Prism. This goes hand in hand with their use of probabilistic model checking for verification of their system. What is more, unlike the goal of this work, they did not implement the verified swarm model in a physics simulator but instead relied on statistical simulation.

The last paper reviewed as background research for this Thesis is "Verified ROS-Based Deployment of Platform-Independent Control Systems" [28]. The authors of the paper focus on low level robotic system performance verification. They attempted to verify that internal system command messages (control code) in a robot are correctly delivered to the corresponding sensors or actuators in the robot. In order to verify that the system behaves as expected they created a tool to automatically create "glue code"

(wrappers for platform-independent component implementation) from the architectural model of the target system. This architectural model is specified using a domain-specific language called ROS node model, which specifies the ROS nodes and topics that make up the robotic system.

As a case study, the paper outlines the use of ROS where they prove that the code generated by their ROS-Gen tool correctly connects the system controller functions to sensors and actuators. Their code generator is implemented using the Coq formal proof management system, the CompCert compiler verifier, and the Verified Software Toolchain (VST) tools. The robotic system used as a case study in this work is a cruise control system for the LandShark robot. The control system architecture for this robot can be seen in Figure 2.10.

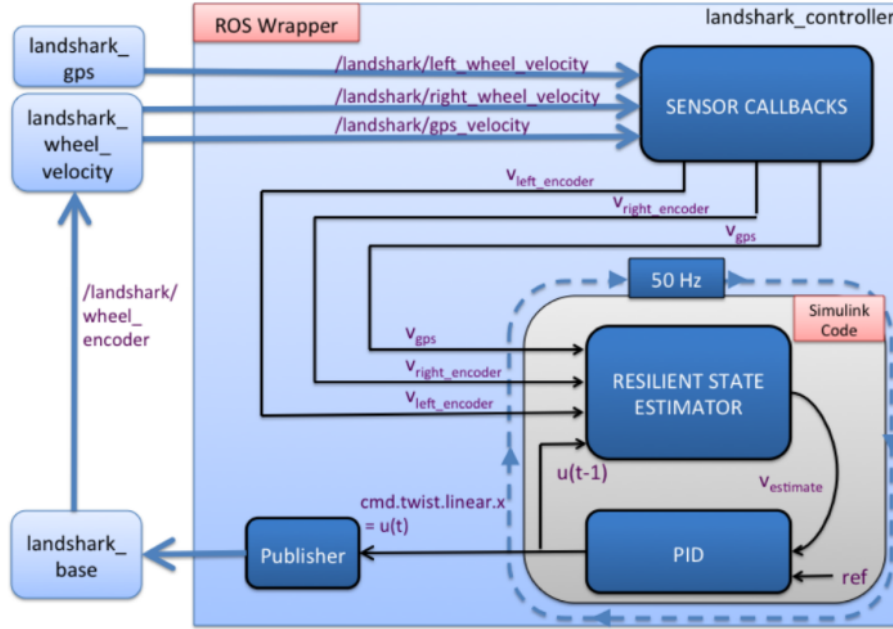


Figure 2.10: LandShark Robot Control System Architecture [28]

The outline of the code generation process is seen on Figure 2.11. The process begins with the modeling of the ROS system definition using the ROSLab modeling

tool. This tool lets users represent a target system architecture using block diagrams, which can then be exported as a ROS node model as introduced above. The model is then used as an input to the tool created by the research group, ROSGen, which produces an abstract syntax tree for Clight (subset of C) by instantiating a Clight AST template. Their tool also generates a VST specification that describes the Data Delivery Correctness (DDC) properties. Finally, the C code that runs on the LandShark robot is created by the CompCert compiler.

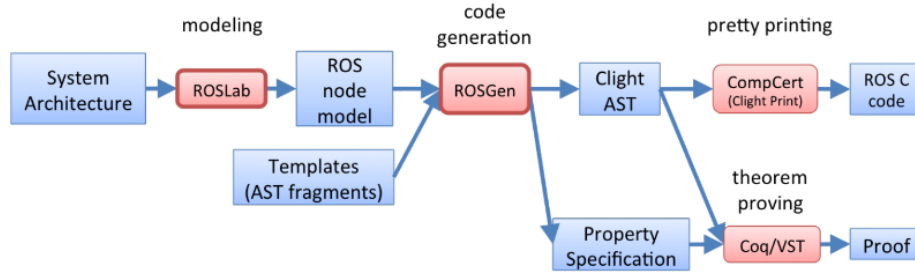


Figure 2.11: Code Generation Process and Tools [28]

One of the most important DDC properties of the generated code verified by the outlined approach is that the message going from sensors to the control functions are correctly delivered, and that the output of the control system functions are correctly delivered to the actuators.

The paper summarized above differs from the goals of this Thesis because they focus on lower level verification of internal communication parameters. Their approach does not include the use of the UPPAAL model checker, and it does not involve the direct translation of a verified model to ROS. However, it does involve some automation in going from a diagram of a system to implementable C code.

Chapter 3

Research Methodology

The following Sections provide a summary of the framework implemented in this work to model, verify, and simulate a swarm of UAVs. The formal model is also briefly explained alongside the assumptions considered in the modeling process.

3.1 Problem Statement Elaboration

The framework introduced in this work facilitates the path that developers take when going from a formal model to a low-level implementation. The practice of modelling and verifying software systems prior to implementation has been proved to be significantly beneficial in cost and error reduction, as introduced in Chapter 1. However, the process is rarely used by development groups because time is typically scarce. Although the benefits of modeling and verifying a software system usually justify the time investment, it is rarely seen in today's technology scene. The work that follows shows a direct mapping algorithm to implement a system modeled in UPPAAL, in a ROS environment. The use of this method aims to reduce the burden and technical challenges faced by developers when considering formal verification prior to low level

implementation of robotic systems.

3.2 Framework

As introduced in previous chapters, there are two principal tools or environments that will be used in this work: UPPAAL and ROS. Concise definitions of these two software environments are provided in section 1.3. The goal is to combine them into a single process where a verified UPPAAL model can be fluently implemented in ROS based on a direct mapping. The representation of the drones' flight dynamics was achieved using an open source Matlab code base developed by a research team from the Brigham Young University. The authors published their work in the paper "UAV Path-Planning using Bezier Curves and a Receding Horizon Approach" [16]. The output of the Matlab script was used as an input global variable in the UPPAAL model. The purpose of including the drone flight dynamics as calculated from the Matlab script is to make up for the abstractions necessary for the formal model.

The UAV swarm model with its logical and physical interactions was developed in UPPAAL. The system represents the behaviour of each drone as well as the commanding role of a Mission Controller. The tool was also used to elaborate and verify properties that prove the correctness of the system. The model was developed using UPPAAL 4.1.24 on a Windows machine. As introduced earlier, the translation of the formal model to ROS was partially automated using an open source GitHub project named `uppaal2ros` [33]. The manual part of the translation, which encompasses the ROS nodes, targeted the representation of the timed automata logic with easily traceable states. In order to visualize the ROS implementation, the system was simulated using Ardupilot and Gazebo. The simulation environment was set up in an Ubuntu 20.04 virtual machine with 12GB RAM, 6 processors, and 256 MB VRAM. It

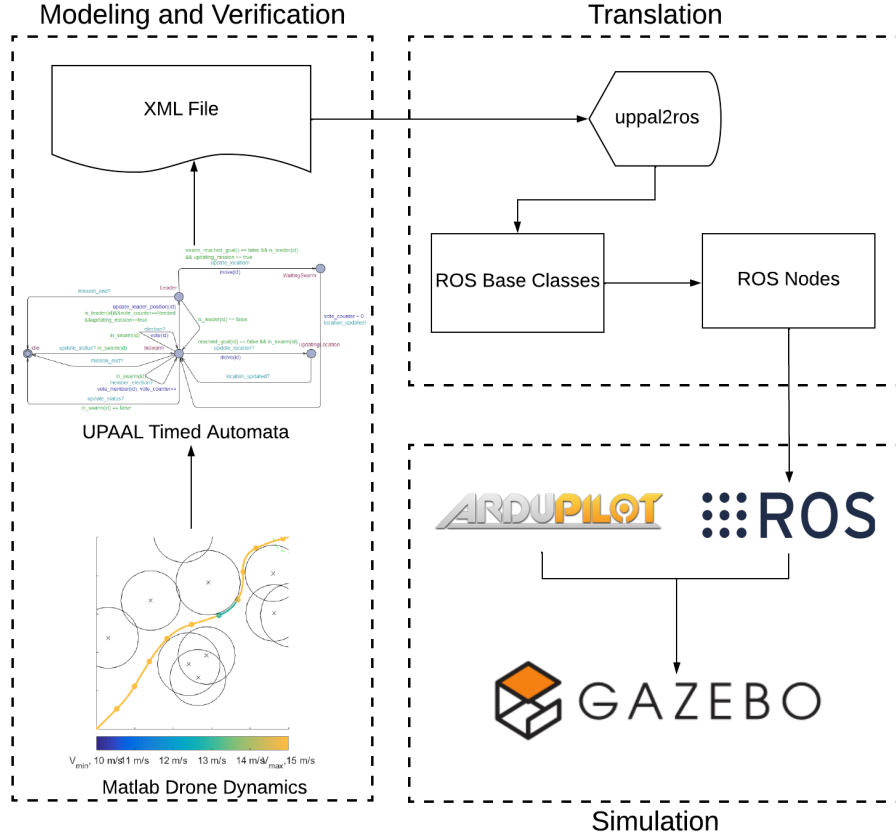


Figure 3.1: Framework Diagram

was hosted in a Windows 10 system with 16 GB RAM, and Intel i7-3770 processor, and a GTX 1060TI dedicated graphics unit with 6GB VRAM. The entire framework is summarized by the diagram shown in Figure 3.1.

3.3 Formal Model

In addition to the requirements outlined in section 1.2, there are numerous assumptions and system design decisions that impacted the modelling of the system. As introduced earlier, the scenario explored in this work is one where a swarm of drones is tasked with extinguishing a wildfire. The location of the wildfire, represented as a red star on

Figure 3.2, is referred to as the "goal location" and it is arbitrarily chosen before the mission starts. The mission described here also involves a Ground Station entity which is mainly tasked with communication. The drones that initially partake in the mission are located near the Ground Station which means they are in the Ground Station Range and they are aware of the goal location for the mission. As it was explained in Section 1.2, these drones must possess the capabilities that are required for the mission as dictated by the Ground Station. Some of the drone agents that will be considered for the mission are initially located further away from the Ground Station and closer to the fire location. These last group of drones which standby along the path of the swarm, are not initially part of the mission but rather join the mission later if they meet the requirements. In other words, the only drones that can initially join the mission are the ones in range of the Ground Station.

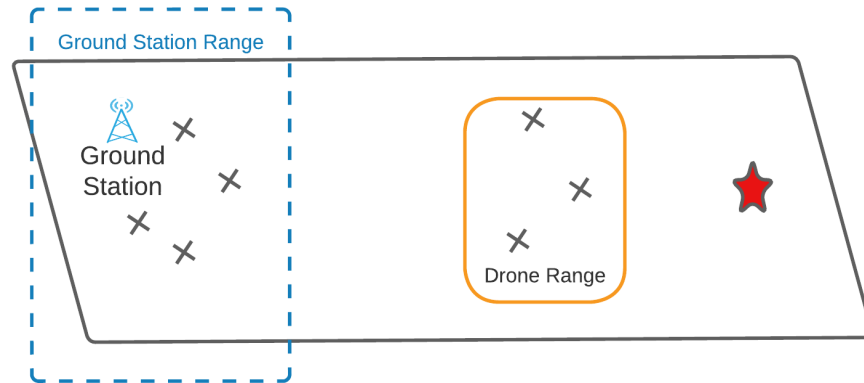


Figure 3.2: Initial Mission Scenario

As depicted in Figure 3.2, not only the Ground Station has a defined communication envelop but so does the leader drone once the mission is at an advanced state. This range means that only drones within the range of the leader can join the swarm or have the goal location relayed to them.

The mission begins with all drones idling on the ground. Three drones, located near the ground station, are initialized to be part of the swarm. Once the group is formed, the consensus algorithm gets triggered to elect a leader agent. The elected leader must have the capability of communication, which is crucial to be able to relay critical mission information once the swarm moves out of Ground Station Range. Any agents that join the swarm later in the mission must not only match the capabilities initially stated, but also have better battery status than the drone with the same capability that is currently part of the swarm. All drones that join the swarm anywhere along the path are also elected through a consensus voting system and the drone that had the same capability gets dropped from the swarm and returns to the idle state.

The movement of the swarm is led by the leader agent. The drone voted as a leader is always the one that moves first, then triggering the movement of the rest of the swarm. What is more, the drones move in a rectangular fashion, first aligning with the target location on one axis and then starting to move in the remaining direction. The trajectory style is represented in Figure 3.3. After every movement cycle, the surroundings are checked for agents that could join the mission. The election processes involve each drone casting a vote and then comparing all the votes for consensus to elect a new member or new leader.

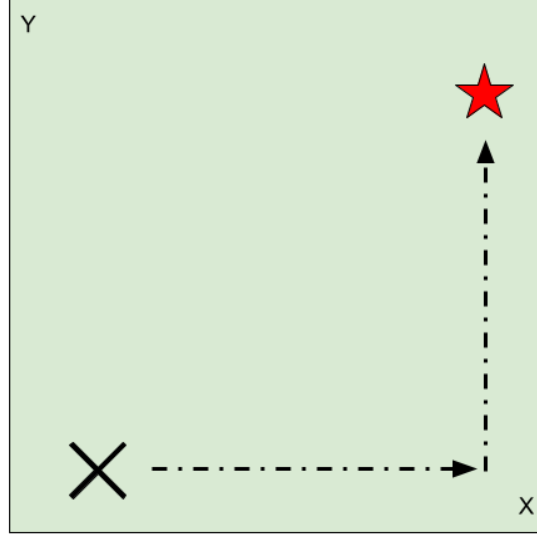


Figure 3.3: Drone Trajectory

3.4 Assumptions

There are a number of assumptions regarding the system model that need to be explained before going into specifics. The total number of drones available in the mission is eight, and only three of these are actively flying as part of the swarm at any point during the mission. With respect to drone dynamics, their movement is defined in two dimensions. Their position is defined by a set of x and y coordinates, while it is assumed that they are flying at a certain height above the ground. This is also true for the location of the fire to be extinguished by the swarm. The location of the ground station is abstracted and assumed to be the same as one of the initial swarm members. The end of the mission, and therefore the end of system execution, is given by the swarm arriving at the fire location. Another assumption regarding the mission is that there is only one drone of each required capability participating in the swarm at any given time.

3.5 Translation

The translation of the UPPAAL swarm model to ROS is partially automated by using the code generated from the work published in [34] and available in [33]. As explained in Chapter 2, the code generator takes an UPPAAL XML file as input and outputs automatically generated ROS compatible files in C++. Specifically, the cited work generates a YAML file with all the UPPAAL global declarations, a ROS launch file instantiating all the nodes corresponding to the UPPAAL system, and a base class that contains all the local declarations for each template. The translation achieved by the authors included a manual process for representing the timed automata logic in ROS. The limitation in their work is that the states in the timed automata are not easily mapped in ROS. Moreover, the implementation does not take full advantage of ROS topics and the publisher subscriber architecture to represent UPPAAL synchronization channels.

The work herein uses the automatically generated files mentioned above, while extending the representation of the TA logic in ROS by addressing the outlined limitations. The translation in this work maps states in UPPAAL by using a switch statement and cases. The manual part of the ROS implementation also introduces the use of publishers and subscribers by representing UPPAAL synchronization channels as topics. The ROS implementation of the swarm model developed in this work is explained in detail in Chapter 5.

Chapter 4

Technical Specifications

The following sections present the technical details of each of the system components introduced in Chapter 3. The Matlab script used as an input to the UPPAAL model to represent the autonomous agent's flight dynamics is explained line by line alongside the modifications introduced as part of this work in Sections 4.1 to 4.1.2. The basic elements and structures of the UPPAAL modeling tool are also introduced to facilitate the understanding of the formal model. The global variables, templates (timed automata), properties verified and underlying code for the consensus protocol are all explained in detail in Sections 4.2 to 4.6.

4.1 Matlab Drone Dynamics

As introduced in [3], the individual drone flight dynamics were calculated in Matlab using an existing project developed by Ingersoll et al. The code developed in [16] is capable of calculating 2D UAV flight trajectories around static or dynamic obstacles of different sizes. Simply stated, the program outputs an optimized trajectory for a drone to travel from an initial location to a final location while avoiding obstacles randomly

located in the path. This program fits the scenario represented in this work because individual drone agents might encounter obstacles such as trees when trying to get from point A to point B in a wildfire mission. The Matlab code was implemented by adding a script that automates the variation of input parameters and records the desired output for a given trajectory.

4.1.1 Inputs and Outputs

There are two types of output for the Matlab script, one of them is a visual representation of the trajectory the drone traces, and the other one is the numerical result for time of flight, energy consumed, and distance travelled. The figure representing the trajectory of the drone from a starting point to a desired target location also shows the locations of the randomly sized and randomly placed obstacles that the drone avoids. What is more, the path traced is color coded to represent the speed of the drone at each point in the path. The nature of the path that the drone takes is determined by the objective function chosen by the user. This is one of the most important input parameters for the script, which decides if the distance traveled, the energy spent, or the time of flight should be optimized. A sample drone trajectory as described above can be seen in Figure 4.1. The reason why most of the trajectory is colored in yellow (which corresponds to the maximum speed of the vehicle) is that the algorithm is set to optimize time. As such, the drone flies as fast as possible wherever possible. It can be seen that in one of the turns around an obstacle, the velocity is decreased and the path is colored green.

The input parameters for the Matlab script are seen in Appendix A. The global variables used in the code are defined in lines 20 to 45. Lines 47 to 66 contain the variables for setting the algorithm options. For example, line 50 is used to define if the trajectory optimization should employ a genetic or evolutionary type of algorithm.

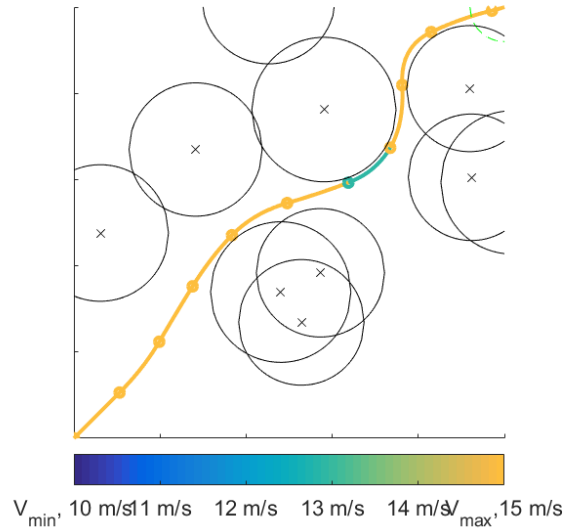


Figure 4.1: Sample Drone Trajectory Optimizing Time [16]

Line 52 lets the user include dynamic (moving) obstacles in the drone path scenario. In this application of the scripts both of these variables were set to 0, which means these features were not used. The most important set of parameters in these lines of codes are the configuration of the objective function in lines 60 and 61. If the variable `optimize_energy_use` on line 60 is set to 1, then the trajectory is optimized for the lowest energy use possible. Instead, if the variable `optimize_time` on line 61 is set to 1, then the trajectory is optimized to give the fastest travel time. Finally, if both of these variables are left to be 0, then the distance traveled by the drone is minimized.

Lines 66 to 102 on Appendix A contain settings for the output plot showing the optimized trajectory. The visualized colors, line width, thickness, and some physical parameters like UAV limit of sight can be modified.

Further along the code, between lines 115 and 185, a number of physical drone parameters are defined. For example, line 119 defines the weight of the UAV and line 121 defines the Oswald aerodynamic efficiency factor. Line 160 sets the vehicle's

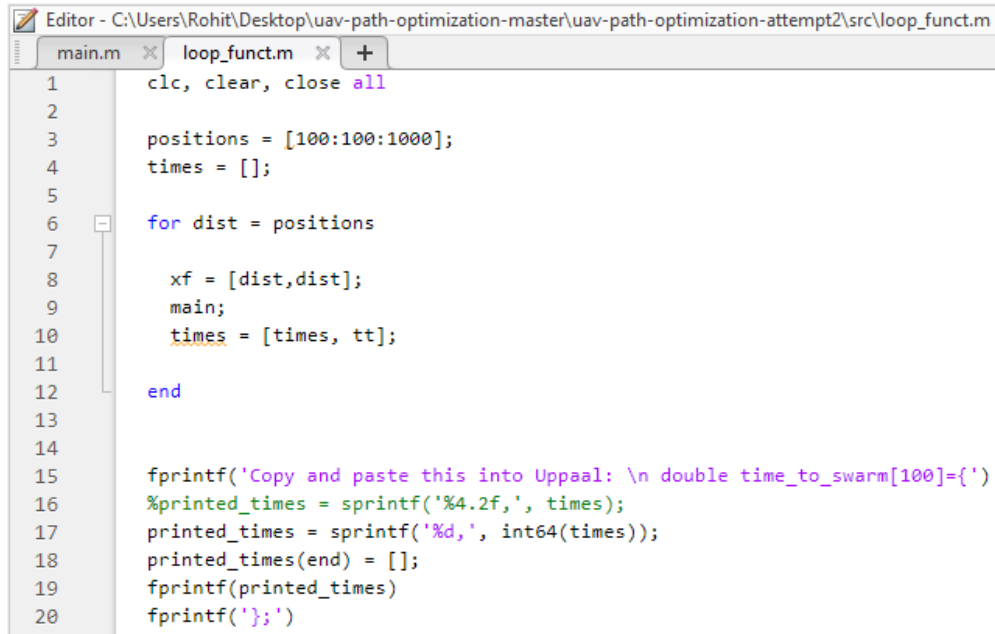
turn radius, while lines 163 and 164 set the UAV's maximum and minimum speeds, respectively. Line 174 establishes the wingspan of the vehicle modeled in the script. The drone's starting position for the trajectory is set in line 181 of the Matlab script. The final position that the drone must get to for the trajectory to be complete would be declared on line 182. However, in the implementation used for this work, this line was commented out. The reason for this is that the script is run on a loop iterating over increasing final positions. This allows us to retrieve the time the drone takes to travel to several different locations. On line 184, the script defines the radius of the landing zone, which must be less than or equal to 15.

Lines 187 to 205 of the Matlab script shown on Appendix A define the characteristics for the static obstacles that are spawned in the field. On line 199, one of the modifications made for the purposes of this study is that the number of obstacles spawned on each run of the script is proportional to the magnitude of the drone's target location. This is to represent the situation where the further a drone travels in a fire rescue mission, the more obstacles it might encounter in its path. Line 200 of the script randomly selects coordinates for the static obstacles in the trajectory. Another modification was added here to correlate the obstacle locations to the set final destination. This was done to be able to spawn obstacles all along the path of the drone while the target destination increases in magnitude. Another randomized quantity is programmed on line 204, which represents the size of the obstacles spawned. This allows to have different sizes of obstacles about the trajectory.

The trajectory calculation algorithms are implemented between lines 236 and 445. The final output plot for the trajectories calculated are introduced on line 446. Finally, the comparison between the trajectories optimized for time, distance, and energy is output through line 499 which calls another script called `compare_of`. The time for the trajectories is retrieved from this script and saved for use in UPPAAL.

4.1.2 Matlab Script Modifications

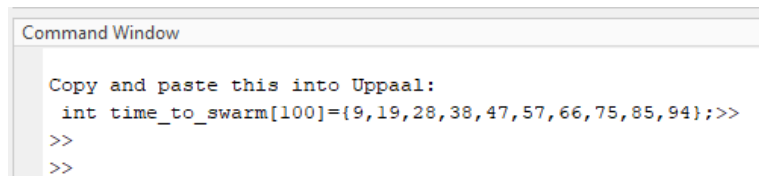
The script included in this section was added to expand the existing Matlab code base developed by Ingersoll et al. The purpose of the file added is to automate the running of the script so that on each run the trajectory's final position is incremented. The script is shown on Figure 4.2 below.



```
Editor - C:\Users\Rohit\Desktop\uav-path-optimization-master\uav-path-optimization-attempt2\src\loop_funct.m
main.m x loop_funct.m x +
1      clc, clear, close all
2
3      positions = [100:100:1000];
4      times = [];
5
6      for dist = positions
7
8          xf = [dist,dist];
9          main;
10         times = [times, tt];
11
12     end
13
14
15     fprintf('Copy and paste this into Uppaal: \n double time_to_swarm[100]={')
16     %printed_times = sprintf('%4.2f,', times);
17     printed_times = sprintf('%d,', int64(times));
18     printed_times(end) = [];
19     fprintf(printed_times)
20     fprintf('');')
```

Figure 4.2: Script Added to Matlab Code-base

Simultaneously, the script saves the time of flight for each trajectory in an array. The data in this array is printed in a statement, as seen on Figure 4.3, that can be copied and pasted into the UPPAAL model before running it.



```
Command Window
Copy and paste this into Uppaal:
int time_to_swarm[100]={9,19,28,38,47,57,66,75,85,94};>>
>>
>>
```

Figure 4.3: Time of flight for different trajectories

4.2 UPPAAL Description

As introduced in Section 1.3, UPPAAL is a tool used to model real-time systems as networks of timed automata extended with data types, with the possibility of then validating and verifying the system behaviour. The tool is used in this work because it is appropriate for systems that can be represented as a collection of non-deterministic processes with finite control structure communicating through channels or shared variables [21]. The version of UPPAAL used in this work was 4.1.24. This section aims to provide basic background knowledge to understand UPPAAL models.

Figure 4.4 shows the file structure for UPPAAL projects. They are composed of Global Declarations, one or more Templates, and System Declarations. Each Template is made up of two parts: a timed automaton and local declarations. The timed automaton is represented as a graph with a set of states and transitions, while the local declarations are essentially a file with variable and function definitions. System Declarations can be thought of as another file in the working tree where the composition of the system is defined. This involves instantiating each of the Templates developed, allowing to pass any required arguments, as well as defining which objects will compose the entire system. An example of System Declarations is explained in Section 4.4.

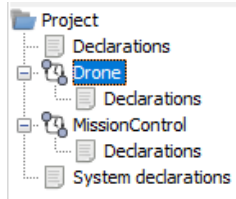


Figure 4.4: UPPAAL Project Structure

Global declarations, as the name suggests, are globally accessible by all templates in a system. These can read and write global variables at any stage of execution. On the other hand, local declarations are tied to individual templates. The functions and

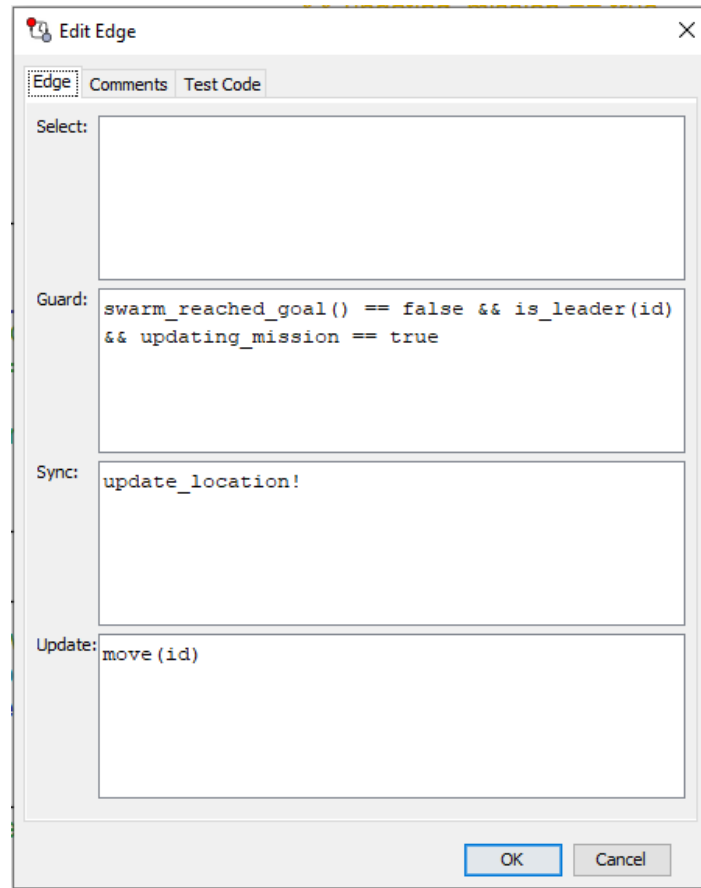


Figure 4.5: Edge Annotations

variables defined within are only accessible by the template they are attached to. In this work, all variables are defined as global in the global declarations section while all functions are defined in each template's local declarations. Finally, variables passed as parameters to the templates in the system declarations can be used within templates as well.

Timed automata are represented as a graph composed of states and edges. States are seen as blue circles with red font labeling. Initial states can be identified by states with two concentric circles. Edges or transitions are shown as arrows going from state to state or looping back to the same state. The logic of the transitions taken through the graph lies on the edges themselves which, as shown on Figure 4.5, can be populated

with Selections, Guards, Synchronisation, and Updates.

According to the UPPAAL Language Reference guide, selections non-deterministically bind a given identifier to a value in a given range [21]. Guards are conditions that enable transitions from one state to another if and only if they evaluate to true. Synchronisation channels are the mechanism through which processes synchronize. Practically these are used to trigger actions or transitions in another template or process. If the synchronization variable is accompanied by an exclamation sign then it sends a signal to all edges labeled with the same variable but accompanied with a question mark. Finally, updates are a consequence of taking the transition where the expression in them gets evaluated. The system developed in this thesis does not include the use of Select edge annotations.

Another of the main features included in the UPPAAL tool is the Verifier interface. It is used to check properties of a system that can be related to safety, liveness, reachability, deadlock, etc. The method it uses to verify if a certain property is satisfied or not, is state-space exploration. The Verifier is ideal for specifying, verifying, and documenting system requirements for traceability purposes. UPPAAL uses the requirement specification language to allow the user to input properties. This is further explained in Section 4.6.

4.3 UPPAAL Global Declarations

Global variable definitions are extensively used in this work to represent different drone qualities and states. The workings of the system developed are based on accessing and modifying different global arrays through functions that define the swarm behaviour. Lines 4 to 123 of the XML file presented in Appendix B show all the global variables defined as part of the system. These are explained below.

The `comm_reach` variable on line 8 represents the communication range of the drones. Only drones within this range from the swarm leader can join the swarm. The `time_to_swarm` array is created based on the output of the Matlab script described in the sections above. The values represent the time for a drone to join the swarm based on how far it is from the leader. The distance is represented by the array indices where the first element represents a distance of 1 meter, and the last element represents a distance of 10 meters (the maximum communication range). Lines 16 and 19 respectively represent the variables that hold the total number of drones and the number of drones needed for the missions. Line 22 is a variable that keeps track of the swarm leader's position. The goal integer variable on line 28 defines the end location for the swarm to achieve its mission. It would represent the location of the fire in the fire rescue mission scenario.

Line 41 defines the `drone_status` array which reflects the status of each drone (according to the array index) at all times during the mission. The status of a drone can be -1, 0, or 1 which corresponds to in swarm, not in swarm, and leader respectively. In a similar fashion, the arrays defined on lines 44, 54, 55, and 67 represent drone parameters like battery level, location, and capabilities.

The arrays and variables defined on lines 48, 81, 84, 87, 89, 92, and 95 are all relevant to the consensus protocols for new member and leader elections. For example, the `drone_candidates` array represents the drones that meet the requirements to join the mission in lieu of other drones that are currently flying. The arrays named `votes` and `member_votes` represent the votes cast by flying drones to elect new members and leaders.

4.4 UPPAAL Template Descriptions

Figures 4.6 and 4.7 show the UPPAAL templates representing the swarm system developed for this work. The system XML file exported from UPPAAL is provided in Appendix B. The system is defined as outlined in Figure 4.8, which shows that there are 8 Drone template instances and a single MissionControl template instantiation.

The Drone template shown in Figure 4.6 is instantiated 8 times which means that the system is composed of 8 drones each of which can be at any of the states seen on the template at any time. The timed automaton's initial location is identified by the double lines around the state "Idle". At this state, the drone in question is on the ground and does not participate in the mission. As described in Section 3.4, only three drones fly as part of the swarm at a time which means that most of the drones will remain at the "Idle" state for the larger part of system execution.

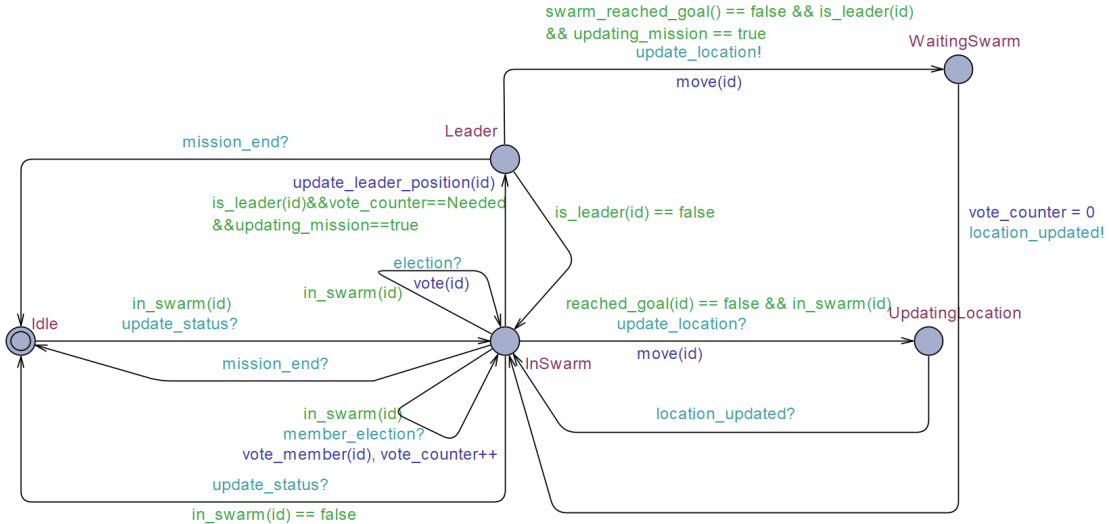


Figure 4.6: UPPAAL Drone Template

As seen on Figure 4.6, only those drones that are part of the swarm move from the Idle state to the InSwarm state. This check is represented by the guard condition on the corresponding edge, which calls the function `in_swarm` seen on lines 138 to 140 of

the XML file presented in Appendix B. The function takes a drone id as input and returns its status as described by the global array `drone_status`.

Once the three drones that partake in the mission area at the InSwarm state, there are two election processes triggered. One of the elections pertains to choosing a leader for the swarm while the other one checks if there are drones in range that would be a better fit for the mission. These two election processes are represented by the two edges that loop back to the InSwarm state. The functions that trigger the elections are `vote` and `vote_member`, both of which can be seen on Figure 4.6. As explained in Chapter 3, three drones are initialized as being part of the swarm, which means that in the first iteration of the execution there is no member election held. The two election protocols are further detailed in Section 4.5 below.

After the two elections are conducted, the mission will proceed with two drones in the "in swarm" status and one drone in the "leader" status. The drone elected as leader will take the transition from the state InSwarm to the state Leader. The function that enables the transition of the leader drone to the Leader state are `is_leader`, which is seen on lines 133 to 135 of the XML file included in Appendix B. Next, the leader drone goes through the move loop which involves the transition to the WaitingSwarm state and back to the InSwarm state. Similarly, the drones in the swarm that are not leaders go through the smaller move loop which involves the UpdatingLocation state and back to the InSwarm state. The move loop transitions are only taken by drones if the swarm has not arrived at the goal location which is verified through the function `swarm_reached_goal` defined on lines 157 to 168 of the UPPAAL XML file. All drones move by calling the `move` function which is defined in lines 211 to 226 of the XML file. At this point the process starts again where an election for new members and an election for leader is held, and then the drones move once more. Whenever a drone is dropped from the swarm during the mission, it takes the transition from InSwarm to

the Idle state. Finally, when any drone reaches the mission goal location, it takes the transition from InSwarm or Leader states to the Idle state.

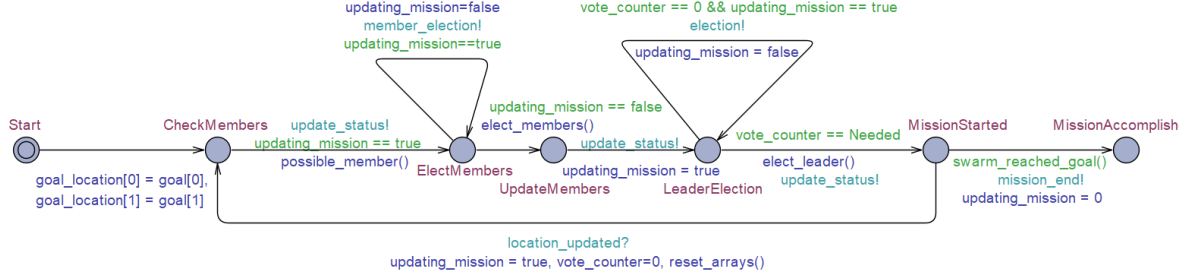


Figure 4.7: UPPAAL MissionControl Template

The second UPPAAL template created as part of the Swarm system is the Mission-Control template shown in Figure 4.7. This timed automata represents a controller entity that manages missions states and triggers drone actions. The template execution begins at the start state labeled Start and shown with the two concentric circles. The first transition is not guarded by any conditions but it does update the goal location variable. The purpose of this is to represent the mission control entity relaying the information for the drones to learn where the fire is located. The next state for the controller is CheckMembers, where the controller checks if there are any members near the swarm that are potential candidates to join the mission. The transition triggers the function possible_member, defined on lines 495 to 506 of Appendix B, which populates the global array drone_candidates as explained in Section 4.3.

At the ElectMembers state the template takes a self loop to trigger the member election process in the swarm. This is achieved through the synchronization channel named member_election!. The next edge towards the UpdateMembers state calls the function elect_members, seen on lines 542 to 570 of Appendix B. This function determines the outcome of the member election by replacing an existing drone with the newly voted member. The next transition simply synchronizes with the Drone

template to update the status of the drones that were added and removed from the swarm.

In the same manner as in the `ElectMember` state, once the mission is at the `Leader-Election` state the self loop commands the drones to elect a leader for the swarm. After voting, if the `vote_counter` variable equals the required number of votes (equal to the number of drones currently participating in the mission), the next transition is taken. At the same time, the `elect_leader` function declared on lines 521 to 529 of the UPPAAL XML file processes the election results. This means that at the `MissionStarted` state the swarm is composed by a leader drone and two loyal wing-man drones. At this stage there are two possible transitions, one looping back to the `CheckMembers` state, and another one leading to the `MissionAccomplished` state. The former is taken if the drones move closer to the goal and send a signal over the synchronization channel `location_updated`. On the other hand, if the function `swarm_reached_goal` introduced earlier returns true, the mission is completed and the controller proceeds to the `MissionAccomplished` state. This transition also sends a synchronization signal over the `mission_end` channel for the drones to shut down and return to the `Idle` state.

Figure 4.8 shows the UPPAAL system declarations which define how the templates introduced above are instantiated. The code snippet shows 8 instantiations of the `Drone` template and 1 instantiation of the `Mission Control` template. The parameters passed to the `Drone` template are the drone id, its capability, and its battery status. These are used in the `Drone` template as part of guard conditions, and as inputs to functions. The last line in the declarations defines the system as being composed of each of the 8 drones and the mission control entity.

```

// System declarations

//drone instances
Drone0 = Drone(0, drone_capability[0], drone_battery[0]);
Drone1 = Drone(1, drone_capability[1], drone_battery[1]);
Drone2 = Drone(2, drone_capability[2], drone_battery[2]);
Drone3 = Drone(3, drone_capability[3], drone_battery[3]);
Drone4 = Drone(4, drone_capability[4], drone_battery[4]);
Drone5 = Drone(5, drone_capability[5], drone_battery[5]);
Drone6 = Drone(6, drone_capability[6], drone_battery[6]);
Drone7 = Drone(7, drone_capability[7], drone_battery[7]);

//mission instance
MissionC = MissionControl();

//system definition
system MissionC, Drone0, Drone1, Drone2, Drone3, Drone4, Drone5, Drone6, Drone7;

```

Figure 4.8: UPPAAL System Declarations

4.5 Election Protocol

Section 4.4 explains the two different election instances that get triggered repeatedly during the execution of the swarm mission. The first one is the member election process, which is responsible for choosing drones that are more suitable for the mission than the ones currently involved. The second election process is the leader election which concludes with the election of a single leader out of the drones flying the mission. The remaining of this section shows and explains the functions used in both the Drone and MissionControl templates to represent the election protocols.

The first step before the system elects new swarm members is checking for new drone candidates. This is achieved through the function `possible_member()` defined in lines 495 to 506 of Appendix B, which is called over the edge leading to the `ElectMembers` state as shown in Figure 4.7. The function includes drones in the candidate list if they are not part of the swarm, have more than 50 units of battery, are within reach of the swarm, and their capability is needed for the mission. After the function is

executed, the `drone_candidates` array has a 1 in the elements representing drones that are candidates to join the swarm.

Figure 4.9 shows the function that the drones call to cast their vote for new members. The routine finds the drone from the candidates list that would take the least time to arrive at the swarm location according to the flight dynamics defined in Section 4.1.2. Furthermore, if that drone also has better battery status than the drone with the same capability currently in the swarm, then the vote is established.

```
// function for drone to cast vote on member election
void vote_member(int id){

    for (i : int[0,N-1])
    {
        if(drone_candidates[i] == 1)
        {
            int current_drone = find_drone_in_swarm(drone_capability[i]);
            calculate_drone_times(drone_capability[i]);
            min_time = find_min(drone_time, drone_capability[current_drone]);
            if(drone_time[i] == min_time)
            {
                if((drone_battery[current_drone]<50) && drone_battery[i] > drone_battery[current_drone])
                {
                    //vote for drone to be part of the swarm
                    member_votes[id] = i;
                }
            }
        }
    }
}
```

Figure 4.9: Member Election Function

Once all drones submit their votes for new members, the `elect_members` function is called by the `MissionControl` template on the transition towards the `UpdateMembers` state. This function processes the election results by calling the function `voting_results`, shown in Figure 4.11, which makes sure all drones agree on the new members. If the function that processes the votes returns false, which means that consensus was achieved, then the new drone is assigned to the swarm and the replaced drone is dropped.

```

// based on the result of the election, this function assigns a member drone
void elect_members(){

    if(vote_counter == Needed)
    {
        for (i : int[0,Needed-1])
        {
            int cap = mission_capabilities[i];
            id_voter = find_drone_in_swarm(cap);
            check_member_votes[i] = member_votes[id_voter];
        }

        if(voting_results(check_member_votes)==false)
        {
            //get id of drone already in swarm with same capability as candidate to join (id)
            int curr_id = find_drone_in_swarm(drone_capability[member_votes[id_voter]]);
            drone_status[curr_id] = 0;
            drone_status[member_votes[id_voter]] = -1;
        }

        voting_in_prog = false;
    }

    updating_mission = true;
    candidate_counter = 0;
    vote_counter = 0;
    reset_arrays();
}

```

Figure 4.10: Member Election Results Processing

```

//checks the results of the votes for leader to see if consensus was achieved
bool voting_results(int check_votes[Needed]){

    for (i : int[0,Needed-2])
    {
        if (check_votes[i] != check_votes[i + 1])
            return true;
    }
    return false;
}

```

Figure 4.11: Checking for Consensus

Analogous to the method for electing new swarm members, the drones in the swarm elect a leader using three functions. When the drones are in the InSwarm state and

the MissionControl entity is in the LeaderElection state, the controller synchronises with the drone through the election channel. This triggers the drones to take the edge that has the function vote, which is shown below in Figure 4.12. The function allows drones that are in the swarm to vote for any one drone that is also in the swarm and has communication capabilities. This is because, as defined in Chapter 3, the leader drone must have communication capabilities.

```
// function for drone to cast vote for leader election
void vote(int id){

    for (i : int[0,N-1])
    {
        if(drone_status[i] == -1 && drone_capability[i] == 1)
        {
            votes[id]=i;
        }
    }
    vote_counter++;
}
```

Figure 4.12: Leader Election Function

Once all the drones in the swarm vote for a leader and the MissionControl process takes the transition towards the MissionStarted state, the function elect_leader is called. This function, shown in Figure 4.13, checks if all the votes match by calling the voting_results function explained above. If the function returns false, meaning that consensus was achieved, the leader drone is assigned a status of 1 in the drone_status global array.

```

// based on the result of the election, this function assigns a leader drone
void elect_leader(){

    for (i : int[0,Needed-1])
    {
        int cap = mission_capabilities[i];
        id_voter = find_drone_in_swarm(cap);
        check_votes[i]=votes[id_voter];

    }

    if(voting_results(check_votes)==false)
    {
        drone_status[votes[id_voter]] = 1;
    }

    voting_in_prog = false;
    updating_mission = true;

}

```

Figure 4.13: Leader Election Results Processing

4.6 Properties Verified

One of the most powerful features within the UPPAAL model checker is the Verifier. This is the interface where queries or properties about a system's behaviour can be defined and checked exhaustively. In the case that a property is not satisfied, the tool is capable of returning a Trace that shows the system's execution path as a counter example. If the property being verified is satisfied the interface shows a green circle as seen in Figure 4.14, otherwise the circle turns up red. The tool also outputs the time and memory used by the computer to achieve the verification of each property.

The symbolic queries used in the UPPAAL Verifier are based on the requirement specification language. The four main symbols that are used in specifying properties are: "E" which means exists a path, "A" which means for all paths, "[]" which represents all states in a path, and "<>" which represents some state in a path. The semantics of this language are summarized in the following temporal properties which

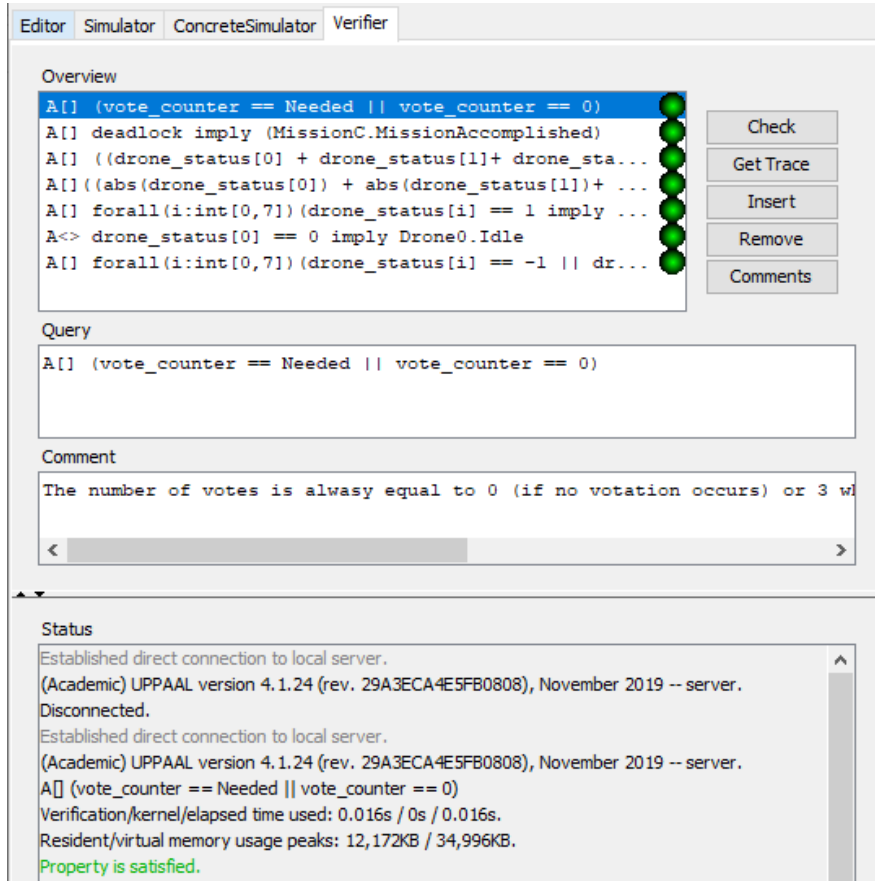


Figure 4.14: UPPAAL Satisfied Properties

assume that p and q are state properties related to the project's Templates containing timed automata.

Possibly: The expression $E<>p$ represents that it is possible to reach a state in which p is satisfied. This means that p is true in at least one reachable state.

Invariantly: The expression $A[] p$ represents that p holds invariantly, it is true in all reachable states.

Inevitably or Eventually: The expression $A<>p$ represents that p will inevitably become true in some state of all paths. The system will eventually reach a state in which p is true.

Potentially Always: The expression $E[] p$ represents that there is a path in

which p holds in all states.

The last construct in the UPPAAL requirement specification language is the symbol “ \rightarrow ” which represents the condition “leads to”. The statement $p \rightarrow q$ means that if p becomes true, then q will inevitably become true. It is equivalent to the expression $A[] (p \text{ imply } A <> q)$.

The properties represented above as p and q can contain logical expressions using any of the global variables defined in the system, any particular state, or the deadlock condition. States are expressed in the form $P.L$ where P is a process (instance of a template) and L is a state in the template’s automaton graph. Furthermore, the property can be defined as deadlock, which would evaluate to true if and only if there is a state where there are no action successors.

The properties verified for the swarm system modeled in UPPAAL are shown below and can also be found in lines 688 to 721 of Appendix B.

1. $A[] (vote_counter == Needed \parallel vote_counter == 0)$
2. $A[] \text{ deadlock imply } (MissionC.MissionAccomplished)$
3. $A[] ((drone_status[0] + drone_status[1] + drone_status[2] + drone_status[3] + drone_status[4] + drone_status[5] + drone_status[6] + drone_status[7]) <= 0)$
4. $A[] ((abs(drone_status[0]) + abs(drone_status[1]) + abs(drone_status[2]) + abs(drone_status[3]) + abs(drone_status[4]) + abs(drone_status[5]) + abs(drone_status[6]) + abs(drone_status[7])) == Needed)$
5. $A[] \text{ forall } (i : int[0, 7]) (drone_status[i] == 1 \text{ imply } drone_capability[i] == 1)$
6. $A <> drone_status[0] == 0 \text{ imply } Drone0.Idle$
7. $A[] \text{ forall } (i : int[0, 7]) (drone_status[i] == -1 \parallel drone_status[i] == 1) \text{ imply } drone_battery[i] >= 49$

Property 1. above verifies that the variable that keeps track of the swarm votes, `vote_counter`, is always equal to 0 or the quantity `Needed`, which represents the number of drones flying in the swarm. The symbols signify that the property holds invariantly in all reachable states.

Property 2. assures that whenever the system is in a state of deadlock, the mission has been accomplished by the drones. This must be true invariantly in all reachable states.

Property 3. corroborates that invariantly, in all reachable states, there is no more than one swarm leader. The logic in the property is derived from the mechanism used to identify the status of the drones. As explained in Section 4.3, the leader is assigned a value of 1 in the `drone_status` global array, while other flying drones get a value of -1. Considering that the current mission is setup to have 3 drones flying at any time, the total of the sum of the elements of the array must always be less than or equal to 0.

Property 4. proves that invariantly, in all reachable states, the number of drones participating in the mission is equal to the value of the variable `Needed`, which represents the drones needed to carry out the mission. The logic is related to the explanation provided above, where the sum of the absolute value of each drone's status must be equal to the value of `Needed`.

Property 5. verifies the correctness of the mechanism used in the election for a leader. The query represents that invariantly in all reachable states, whenever a drone is the leader and therefore has a status of 1, then its capability must be equal to 1 which represents communication. This requirement is laid out in Section 3.

Property 6. assures that whenever a drone is dropped from the swarm and its status returns to 0 in the global array `drone_status`, then it must eventually return to the Idle state in the automaton graph. This must hold true in some state of all the paths.

Property	Verification(s)	Kernel(s)	Elapsed(s)
Property 1	0	0	0.009
Property 2	0.016	0.015	0.025
Property 3	0	0	0.012
Property 4	0.016	0	0.006
Property 5	0	0	0.007
Property 6	0	0	0.002
Property 7	0.015	0	0.013

Table 4.1: Verification Time Metrics

Property 7. is also intended to verify the correctness of the consensus based election mechanisms developed in this work. The statement represents that invariantly, for all reachable states, if a drone is in the swarm either as a leader or as a wing-man, then its battery status must always be larger than or equal to 49.

The Verifier interface includes a Status box, seen on Figure 4.14, which outputs the status of the verification process for each query. This includes the processor time usage which is divided into time used for verification, time used by the kernel, and the elapsed time used [39]. Table 4.1 shows the time metrics for each of the queries explained above.

The most computationally intensive verification was observed for property 2 which took 0.016 seconds of verification time, 0.015 seconds of kernel time usage, and 0.025 seconds of elapsed time. The least demanding property was property 6 which only took 0.002 seconds of elapsed time.

Chapter 5

ROS Implementation

The details of the ROS implementation developed as part of the framework to port the verified UPPAAL swarm model are explained in the following Sections. The main ROS concepts used in the implementation are explained for the reader. Furthermore, the ROS mapping of the UPPAAL structures are presented in tabular and algorithmic formats.

5.1 ROS Concepts

The ROS platform has become the standard choice when developing robotic systems. In 2019 alone it was downloaded more than 264 million times, attesting to its popularity and practicality [4]. Currently, the ecosystem is only supported for Unix-based platforms which is why a Virtual Machine running Ubuntu was used in this work. The ROS distribution used was Noetic. ROS can be described as a robot middleware-layer of software that provides services and capabilities not offered by the OS—typically used as an open source framework to develop robot software [11]. The runtime architecture is a peer-to-peer network of processes that are coupled using the ROS communication

infrastructure [10]. The two inter-process communication methods employed in this work are the asynchronous data streaming over topics, and the data storage provided by the Parameter Server. Furthermore, the language used for development is C++ which is enabled by using the roscpp client library.

The broadest organizational unit in ROS software systems are Packages. They include ROS Nodes, libraries, datasets, configuration files, and other system related materials [11]. The purpose of packages is to provide useful functionality that can be released and easily reused by other developers. Virtually, a Package is a directory that has a package.xml file in it alongside other sub directories containing software.

The ROS Parameter Server is shared, multi-variable dictionary that allows data to be stored in a central location. It is often used by nodes to store system parameters at runtime, while still providing read and write access. The server is not intended to be used for high performance execution, which is why it is typically used to store static data [11]. In this work, the Parameter Server is represented by a YAML file.

ROS Nodes compose the robot control systems, they are the processes that perform computations or tasks. They communicate with each other through Topics or Services where ROS Messages are relayed. The swarm model introduced in this work is composed of multiple identical nodes representing each of the drones, and a node that represents the Mission Controller.

ROS Topics are communication structures where messages are published so they are available to other nodes based on subscriptions. Each topic represents a different type of data, which means that they can have multiple publisher and subscriber Nodes concurrently. The Nodes interacting with topics are not aware of the origin or destination of the data, which means that the process is anonymous.

Messages are the data structures that get published onto ROS Topics. The system developed herein only employs the standard primitive types—integers, float, bool, etc—

while custom defined structures are also acceptable.

Lastly, the ROS Master Node is a special Node that allows Nodes to find each other, exchange messages, and invoke services. It provides naming and registration services for regular nodes, and enables publishers and subscribers to recognize topics.

The interactions and architecture of the ROS concepts introduced above are represented visually in Figure 5.1.

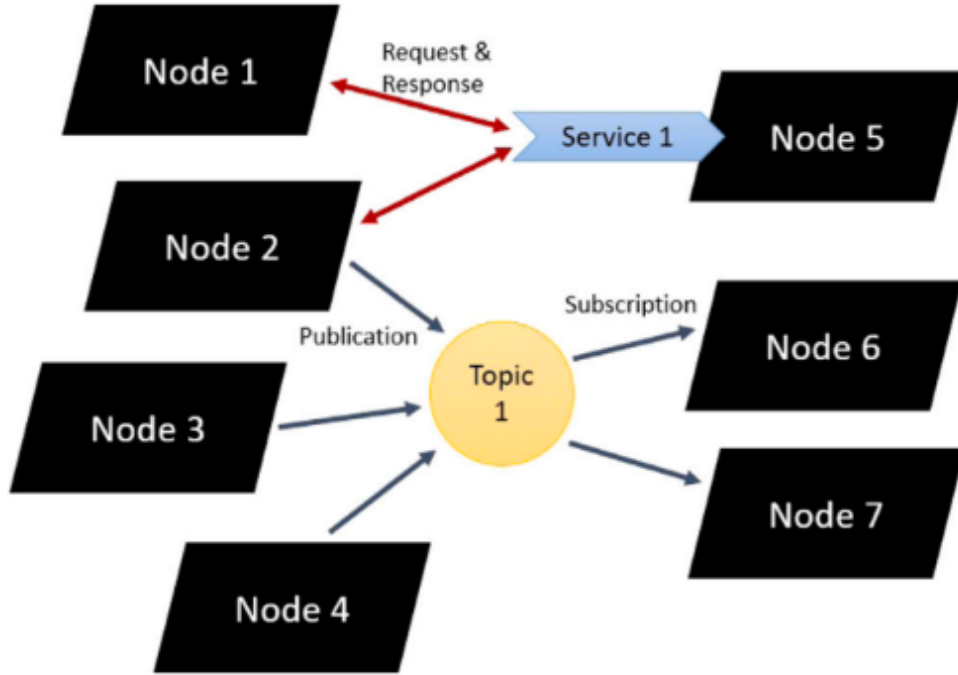


Figure 5.1: ROS Communication Architecture [25]

5.2 Automated Translation

As briefly mentioned in Chapter 3, the ROS implementation of the UPPAAL swarm model was partially automated using the work presented in [34]. The research team published a GitHub open-source project [33] where some of the ROS files needed for

implementation are automatically generated based on an UPPAAL system XML file. The details of the automatic generation process are explained in Chapter 2.

The proposed framework uses the automatically generated parameter server YAML file and the ROS base classes to build a directly mapped ROS system.

Figure 5.2 shows the generated YAML file which can be directly mapped to the UPPAAL Global Declarations described in Section 4.3. As introduced earlier, the UPPAAL swarm model relies heavily on the use of global variables and arrays. This is typically true for any system developed using the UPPAAL model checker. This represents a challenge when porting the system to ROS because the parameter server is not intended to be used for variables that change often. It is merely intended for static variables that define configurations of the system. Nevertheless, the implementation introduced here makes use of the parameter server as it is used in UPPAAL. Therefore, the variables shown in Figure 5.2 are loaded into the ROS nodes initially but they keep changing as the drones execute the mission.

The second set of files used in the ROS representation of the UAV swarm are the generated base classes which correspond to each of the UPPAAL templates' local declarations. These files contain all the local functions used by each template's timed automaton graph in state transitions. For example, Figure 5.3 shows the translation of the vote function defined in lines 230 to 242 of the UPPAAL XML file in Appendix B. We can see that the automated translation leverages the use of the ROS functions `getParam` and `setParam` to update global variable values. The first file is the `Drone_base_class.cpp` and the second file is the `MissionControl_base_class.cpp`, each of which correspond to the UPPAAL templates `Drone` and `MissionControl`. These files are included in Appendix C.

A summary of the translation process, including both the automated and manually elaborated parts, is shown in Table 5.1.

```

! params.yaml X
Translator > manual_translation > src > iq_gnc > global_params > ! params.yaml
1  N: 8
2  Needed: 3
3  candidate_counter: 0
4  check_member_votes: [0, 0, 0]
5  check_votes: [0, 0, 0]
6  comm_reach: 10
7  dist: 0
8  drone_battery: [55, 100, 175, 190, 185, 180, 180, 100]
9  drone_candidates: [0, 0, 0, 0, 0, 0, 0, 0]
10 drone_capability: [1, 1, 3, 2, 2, 3, 5, 1]
11 drone_location_x: [10, 60, 14, 10, 15, 10, 20, 20]
12 drone_location_y: [12, 20, 10, 17, 10, 20, 10, 10]
13 drone_status: [-1, 0, -1, -1, 0, 0, 0, 0]
14 drone_time: [0, 0, 0, 0, 0, 0, 0, 0]
15 election: 0
16 goal: [60, 60]
17 goal_location: [0, 0]
18 id_voter: 0
19 leader_position: [0, 0]
20 location_updated: 0
21 member_election: 0
22 member_votes: [0, 0, 0, 0, 0, 0, 0, 0]
23 min_time: 0
24 mission_capabilities: [1, 2, 3]
25 mission_end: 0
26 reached_goal: 0
27 time_to_swarm: [10, 21, 29, 38, 47, 57, 66, 76, 85, 95]
28 update_location: 0
29 update_status: 0
30 updating_mission: 1
31 vote_counter: 0
32 votes: [100, 100, 100, 100, 100, 100, 100, 100]
33 voting_in_prog: 0

```

Figure 5.2: Automatically Generated ROS Parameter Server YAML File [33]


```

173 void vote(int id)
174 {
175     vector<int> drone_capability;
176     nh.getParam("/drone_capability", drone_capability);
177     int vote_counter;
178     nh.getParam("/vote_counter", vote_counter);
179     int N;
180     nh.getParam("/N", N);
181     vector<int> votes;
182     nh.getParam("/votes", votes);
183     vector<int> drone_status;
184     nh.getParam("/drone_status", drone_status);
185     for (int i = 0; i <= (N - 1); i++)
186     {
187         if ((drone_status[i] == (-1)) && (drone_capability[i] == 1))
188         {
189             votes[id] = 1;
190             nh.setParam("/votes", votes);
191         }
192     }
193     vote_counter++;
194     nh.setParam("/vote_counter", vote_counter);
195 }

```

Figure 5.3: Base Class Voting Function Example

UPPAAL	ROS
Global Declarations	ROS Parameter Server (YAML)
Local Declarations	ROS Base Classes
Timed Automaton	ROS Nodes
Synchronization Channels	ROS Topics
System Declaration	ROS Launch File

Table 5.1: UPPAAL to ROS High-Level Mapping

5.3 Manual Translation

The representation of the logic in the UPPAAL timed automatons was translated to the ROS system manually. The files created as a result correspond to each of the ROS nodes instantiated as part of the system. These files, which can be considered the ones that directly show the behaviour of the drones and mission controller, are Drone.cpp and MissionControl.cpp. The low-level mapping of UPPAAL elements to ROS is summarized in Table 5.2.

UPPAAL	ROS
TA States	Switch Cases
Channels!	Publishers
Channels?	Subscribers
Global Variables	ROS Parameters
Guard Conditions	If Conditions
Updates	Function Calls/Set Parameters

Table 5.2: UPPAAL to ROS Low-Level Mapping

The most straight forward mapping between the two environments was to implement a switch statement where each of the automaton’s states is a case. Inside each case, several nested if statements were introduced to represent the logic and actions in the transitions of the TA. Specifically, the hierarchy of the nested if statements reflects the order in which conditions are checked when a transition is taken in UPPAAL.

Transitions are first enabled by guard conditions followed by a synchronization commands, as defined in Chapter 4. This means that, in ROS, the outer if statement is conditioned by the same condition as the guard which is often a certain value for a global parameter. Within that if statement, the program checks for synchronization. If the UPPAAL edge contains a synchronization channel with a question mark, then it was represented as a subscriber. On the other hand, if the edge contains a synchronization channel with an exclamation mark, then it was programmed as a publisher. The topic that these interact with simply carries an integer message with a 1. Consequently, when the edge that sends the synchronization signal is taken, a 1 is published to the topic where the subscriber node will read it and trigger an action.

In order for processes to wait for synchronization, two different while loops were introduced. At the subscribing end, the program waits to receive the message from the publisher. This is achieved by reading a variable off the callback function. On the publishing end, the program waits for the subscriber to subscribe to the topic. This

was achieved by creating the subscriber within the first if block and shutting it down at the end of it.

Finally, if both the guard and synchronization are met, the functions called in the template's edges as updates are invoked using the base classes introduced above. The files representing the ROS nodes are Drone.cpp and MissionControl.cpp which are presented in Appendix D. The logic followed to map the timed automaton states and transitions to the C++ file representing ROS Nodes is specified in Algorithm 1.

```

1: Global  $Vars \leftarrow \{var_1, var_2, \dots, var_i, \dots, var_m\}$ 
   where  $m$  = number of global variables in TA
2:  $Chan \leftarrow \{chan_1, chan_2, \dots, chan_n\}$ 
   where  $n$  = number of synchronization channels in TA
3: for all  $i \in \{1, \dots, m\}$  do
4:    $PS_i \leftarrow var_i$ ;
5: end for
6: for all  $i \in \{1, \dots, n\}$  do
7:    $PubSub_i \leftarrow chan_i$ ;
8: end for
9:  $PS \leftarrow \{P_1, P_2, \dots, P_m\}$ 
10:  $PubSub \leftarrow \{P/S_1, P/S_2, \dots, P/S_n\}$ 
11: for all  $j \in \{1, \dots, p\}$  do
12:    $N_j \leftarrow TA_j$ ;
   where  $p$  is the number of templates in TA
13:    $L \leftarrow \{loc_1, loc_2, loc_3, \dots, loc_q\}$ 
14:   for all  $k \in \{1, \dots, q\}$  do
15:      $STATES_k \leftarrow loc_k$ 
     where STATE are Enum
16:   end for
17:    $cond_j \leftarrow G_j$ ;
18:   while( $ros::ok()$ )
19:     switch ( $STATE$ )
20:     for all  $STATES$  do
21:       case  $STATE$ 
22:         Edge  $\rightarrow \{\text{if } \{cond_j\} \text{ then}$ 
23:            $PubSub_j \rightarrow publish()/callback()$ 
24:            $A \rightarrow update\{var_j\}$ 
25:         end if}
26:       end if}
27:     end for
28:   end for
29: end for
Algorithm 1: Generate  $ROS = (N, PS, PubSub)$  from  $TA = (L, l_0, Vars, G, Chan, A, Edge)$ 

```

Chapter 6

Validation

The behaviour achieved through the framework to port the UPPAAL system to ROS is validated by means of 3D simulation. The following Sections explain the expected behaviour based on the UPPAAL system, the simulation environment setup, and the results obtained in Gazebo using the ROS implementation.

6.1 Simulation Setup

The simulation environment used to validate the behaviour of the swarm based on the ROS implementation comprises three interfaced components: ROS, Ardupilot, and Gazebo.

Ardupilot is an open-source unmanned vehicle software suite or autopilot that supports UAVs amongst a range of other systems [7]. The ArduPilot firmware works on a variety of hardware units and can be coupled with ground control software to add functionalities like real-time communication with operators [3]. The software can be paired with multiple peripheral devices like sensors, companion computers, and communication systems.

Gazebo is an open-source 3D robotics simulator which integrates real-world physics through various physics engines to provide high fidelity simulations. Much like ROS and ArduPilot it is composed of a collection of software libraries that simplify the development process. The advantage of using Gazebo is that it offers integration with a plethora of sensors and robotics hardware [29].

Software-in-the-loop (SITL or SIL) is a technique to test and validate software that is intended to run on hardware, by creating a virtualized setup and simulating its behaviour. Such simulations are a great way to find software faults without the need to purchase any hardware and worry about physical damage [40]. It is commonly used to simulate software for navigational purposes running on different types of vehicles like cars or UAVs. One of the biggest advantages of using ROS, ArduPilot, and Gazebo is that they congruently support SITL simulation.

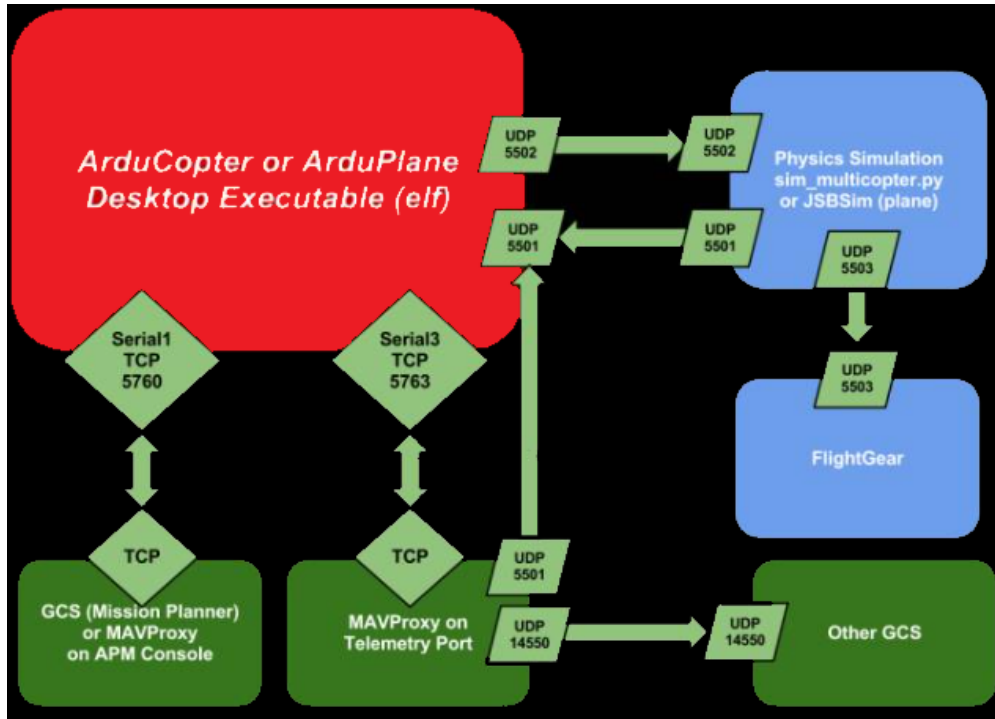


Figure 6.1: ArduPilot SITL Architecture [3]

The ArduPilot project offers a build of its autopilot that gives a native executable SITL simulator. This allows the Copter, Plane, and Rover versions of their vehicle code to be run without any hardware other than a PC. When running a SITL simulation, sensor data is received from a model of the vehicle in a flight simulator, and the autopilot treats it as data coming from a real world vehicle. In this work, the simulator integrated to the SITL simulation is Gazebo. Figure 6.1 shows the interfaces between the autopilot stack (ArduPilot), the vehicle physics simulator (FlightGear or Gazebo), and the ground control station applications (MAVProxy). Although the specific names of the components do not match the ones used in this research effort, it can be seen that an arrangement of UDP and TCP ports are used to establish communication links. Furthermore, this figure does not show the integration of the ROS component.

The diagram in Figure 6.2 shows how the SITL Ardupilot architecture fits in with the ROS middleware through the use of MAVROS. This is a ROS package that enables MAVLink extendable communication between computers running ROS for any MAVLink enabled autopilot, ground station, or peripheral [1].

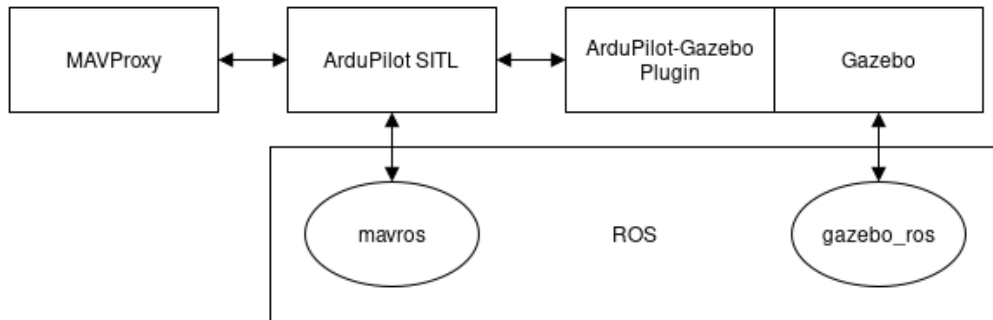


Figure 6.2: SITL Simulation Setup [45]

The Gazebo simulator interfaces with the ArduPilot SITL instance through the ArduPilot-Gazebo Plugin shown in Figure 6.2. Simultaneously, the interface between Gazebo and ROS is facilitated by the gazebo_ros package. The setup introduced above is thoroughly explained in a series of tutorial videos on the YouTube channel "Intelli-

gent Quads” under the playlist named ”Drone Software Development Tutorials” [35]. Furthermore, their work is available on GitHub which has two ROS packages under the repositories ”iq_gnc” and ”iq_sim” [36]. Both of these resources were used extensively in this work to achieve the simulation setup described.

6.2 UPPAAL Symbolic Simulator

The behaviour of the UAV swarm model is best appreciated through the UPPAAL Symbolic Simulator interface. It is a validation tool enabling the examination of the possible dynamic executions of the system developed. It also helps the developer visualize symbolic traces generated by the Verifier introduced in Section 4.6. This means that if a property specified in the Verifier interface is not satisfied, the counter example trace provided by UPPAAL will be visualized in the Simulator. The developer is able to see exactly which path of execution lead to the unsatisfactory state.

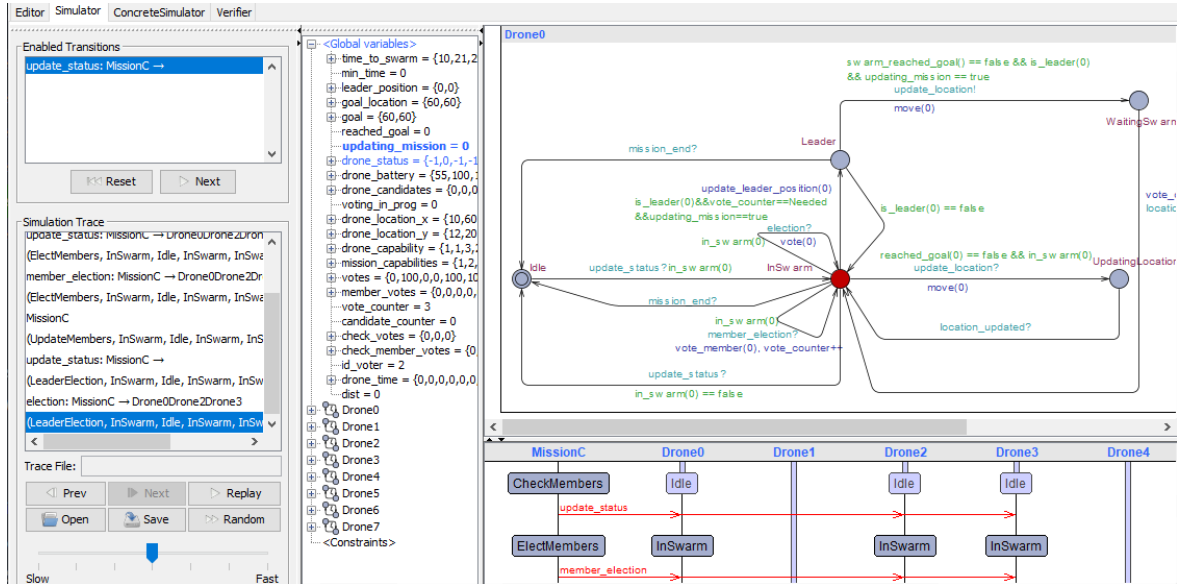


Figure 6.3: UPPAAL Symbolic Simulator Snapshot

Figure 6.3 shows a snapshot of the UPPAAL Symbolic Simulator interface which

is made out of four panels. The leftmost one is the Simulation Control panel where the state or transition desired can be selected and the current trace is displayed. The second panel is Variables Panel where the value of each variable at the selected state or transition is displayed. The top right panel is the Processes window where each instantiated template and its current state is shown. The current control point of each automaton is highlighted in red, as well as the currently enabled transition. Finally, the lower right panel is the Message Sequence Chart where the the current state of each process and the communication between them are shown.

6.3 Simulation Test Cases

The purpose of simulating the UAV swarm system using ROS and Gazebo is to validate the implementation process explained in Chapter 5. The goal of the simulation is to observe the same behaviour reflected by the UPPAAL model. The best way to observe the behaviour of the verified formal model is through the UPPAAL symbolic simulator introduced in Section 6.3. The expected behaviour is described in the following paragraphs.

Initially, the UPPAAL model is loaded with the Global Declarations as shown in Figure 6.4. The first validation task is checking if these are correctly loaded and accessible by the simulated drones in the ROS Parameter Server.

Based on the explanation provided in Chapter 5, the Global Declarations were mapped to the ROS Parameter Server. The variables in the server can be listed by using the command "rosparam list" while the system is running. The output is shown in Figure 6.5.

Regarding the drones, the first scenario to validate is the initial swarm members. Figures 6.6 and 6.7 reflect that the two drones that get initially assigned to the swarm

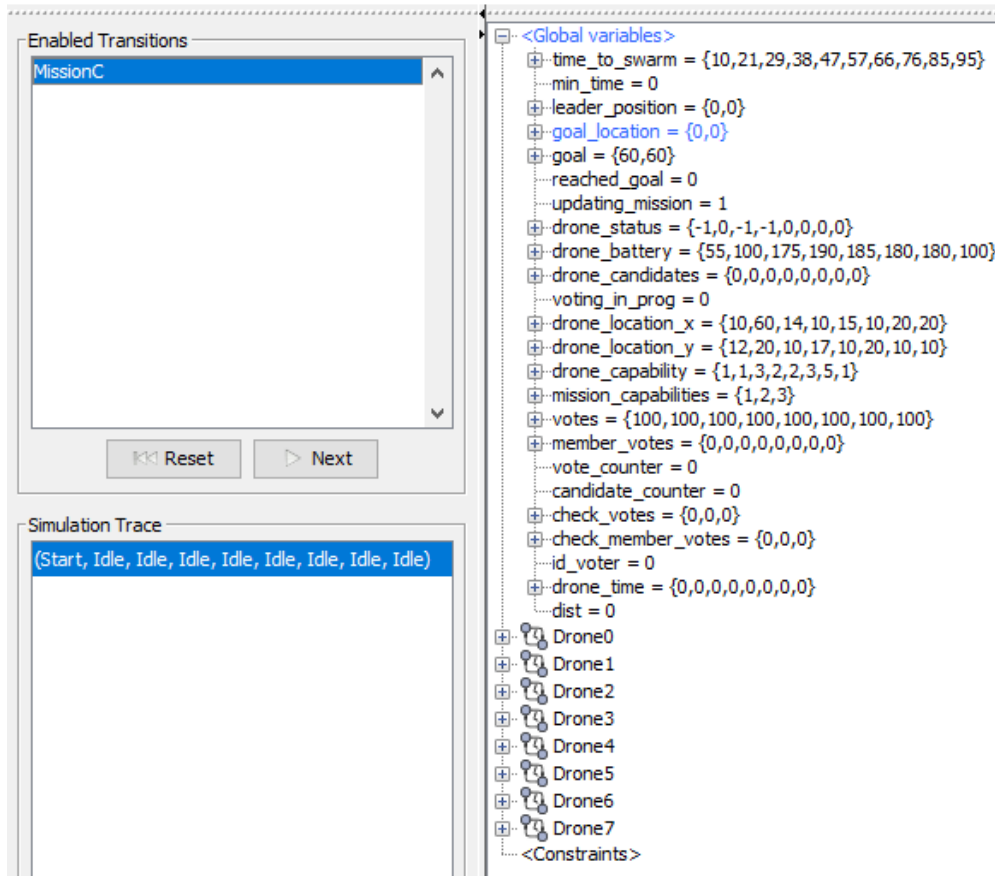


Figure 6.4: UPPAAL Global Parameters

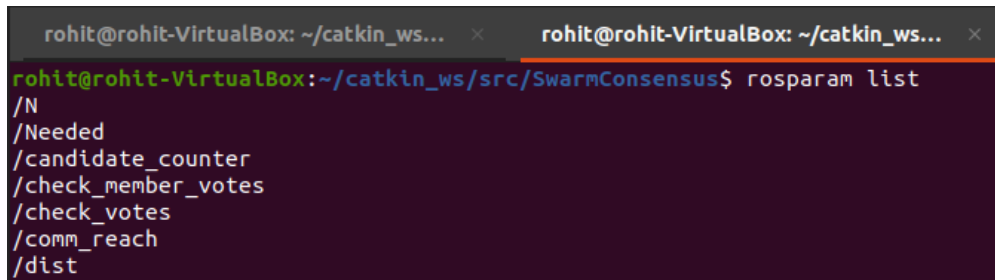


Figure 6.5: ROS Parameter List

are drones number 0, 2, and 3. This is also reflected in the ROS environment where drones 0, 2, and 3 takeoff because they are in the switch case for the InSwarm state. The simulation is shown in Figure 6.8.

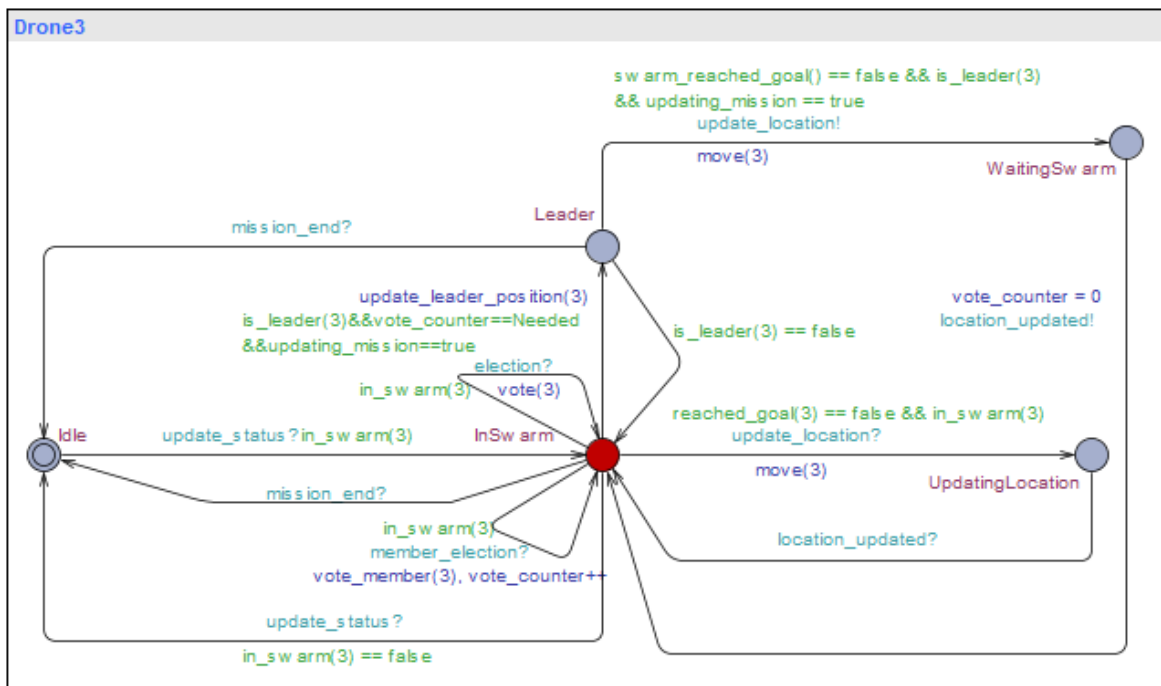


Figure 6.7: Drone 3 Initially In Swarm State

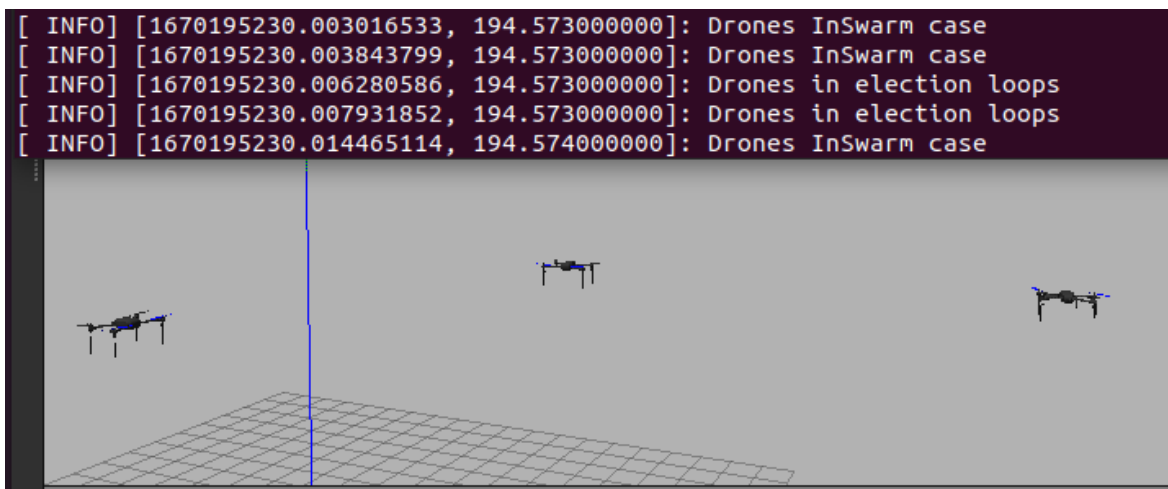


Figure 6.8: ROS Drones In Swarm

members of the swarm agree to drone 0 being the leader.

Figure 6.10 shows that drone 0 is the one that moves to the Leader state. It can also be seen that the transition to move is enabled, showing that the leader always


```

rohit@rohit-VirtualBox:~$ rosparam get /drone_status
- 1
- 0
- -1
- -1
- 0
- 0
- 0
- 0
rohit@rohit-VirtualBox:~$

```

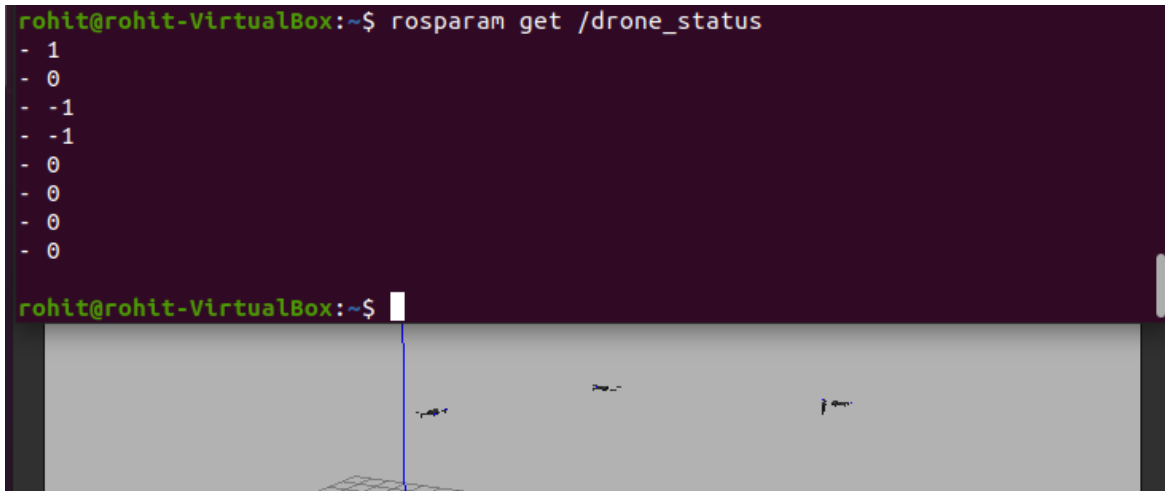


Figure 6.11: Drone 0 Leader in ROS

array shows that element 0 is set to 0 and element 7 is set to -1. This means that drone 0 returns to the Idle state while drone 7 is now in the swarm. Figure 6.13 shows that the timed automaton for drone 0 returns to the Idle state as expected.

```

<Global variables>
+ time_to_swarm = {10,21,29,38,47,57,66,76,85,95}
+ min_time = 38
+ leader_position = {16,12}
+ goal_location = {60,60}
+ goal = {60,60}
+ reached_goal = 0
+ updating_mission = 1
+ drone_status = {0,0,-1,-1,0,0,0,-1}
+ drone_battery = {49,100,169,184,185,180,180,100}
+ drone_candidates = {0,0,0,0,0,0,0,0}
+ voting_in_prog = 0
+ drone_location_x = {16,60,20,16,15,10,20,20}
+ drone_location_y = {12,20,10,17,10,20,10,10}
+ drone_capability = {1,1,3,2,2,3,5,1}
+ mission_capabilities = {1,2,3}
+ votes = {0,100,0,0,100,100,100,100}
+ member_votes = {7,0,7,7,0,0,0,0}
+ vote_counter = 0
+ candidate_counter = 0
+ check_votes = {0,0,0}
+ check_member_votes = {7,7,7}
+ id_voter = 2
+ drone_time = {0,0,0,0,0,0,0,0}
+ dist = 4

```

Figure 6.12: Drone 7 Elected As New Member

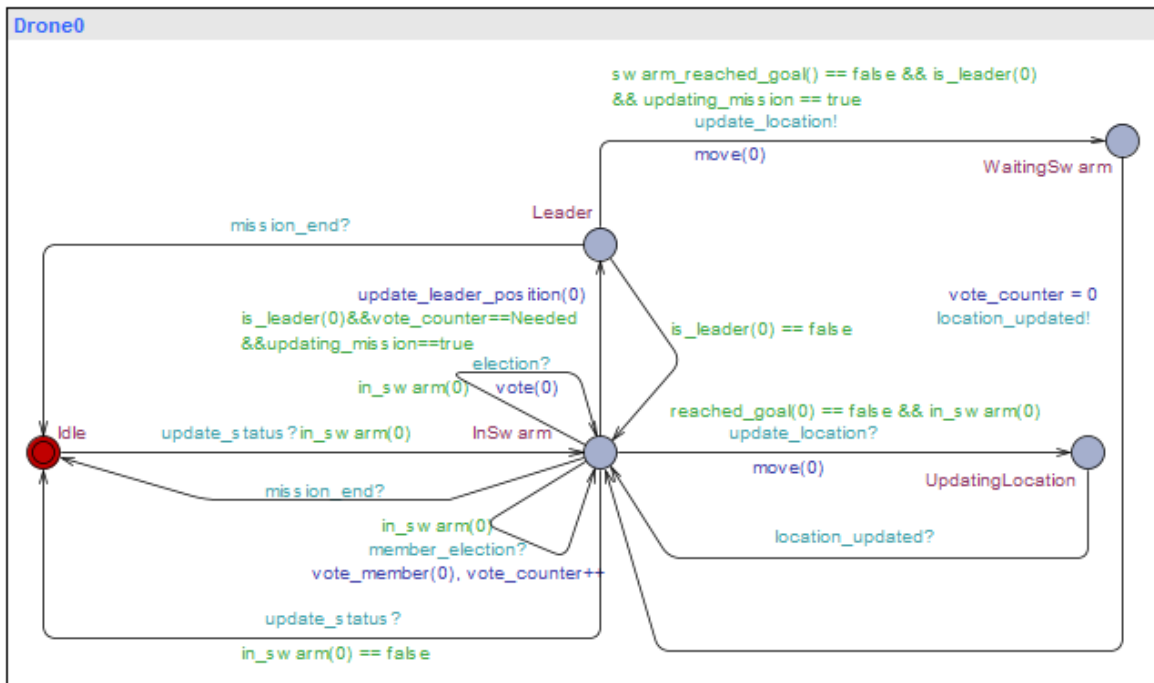


Figure 6.13: Drone 0 Idle State

Following the member election, the swarm elects a leader. In this case, the newly added member, drone 7, gets elected as leader. Figure 6.14 shows that the global arrays are updated to reflect the results of the election. The status of drone 7 is updated to 1. Figure 6.15 shows that drone 7 proceeds to the Leader state.

Then, the drones keep moving through the timed automata loops. Figure 6.16 shows that after several iterations, the leader drone which is still drone 7, line up with the target on the x axis.

Drone 7 below 50 batter gets dropped, drone 1 becomes member Eventually, drone 7 runs low on battery and its level goes below 50. As such, it gets dropped from the swarm and drone 1 joins instead, after being elected as new member. Figure 6.17 shows the drone status array where element 7 is set to 0 again and element 1 is set to -1.

At the next step of system execution, drone 1 is elected leader. This is seen on Figure 6.18 which shows the state of each process in parallel.

```

<Global variables>
+ time_to_swarm = {10,21,29,38,47,57,66,76,85,95}
+ min_time = 38
+ leader_position = {20,10}
+ goal_location = {60,60}
+ goal = {60,60}
+ reached_goal = 0
+ updating_mission = 1
+ drone_status = {0,0,-1,-1,0,0,1}
+ drone_battery = {49,100,169,184,185,180,180,100}
+ drone_candidates = {0,0,0,0,0,0,0}
+ voting_in_prog = 0
+ drone_location_x = {16,60,20,16,15,10,20,20}
+ drone_location_y = {12,20,10,17,10,20,10,10}
+ drone_capability = {1,1,3,2,2,3,5,1}
+ mission_capabilities = {1,2,3}
+ votes = {0,100,7,7,100,100,100,7}
+ member_votes = {7,0,7,7,0,0,0,0}
+ vote_counter = 3
+ candidate_counter = 0
+ check_votes = {7,7,7}
+ check_member_votes = {7,7,7}
+ id_voter = 2
+ drone_time = {0,0,0,0,0,0,0,0}
+ dist = 4

```

Figure 6.14: Drone 7 Elected Leader

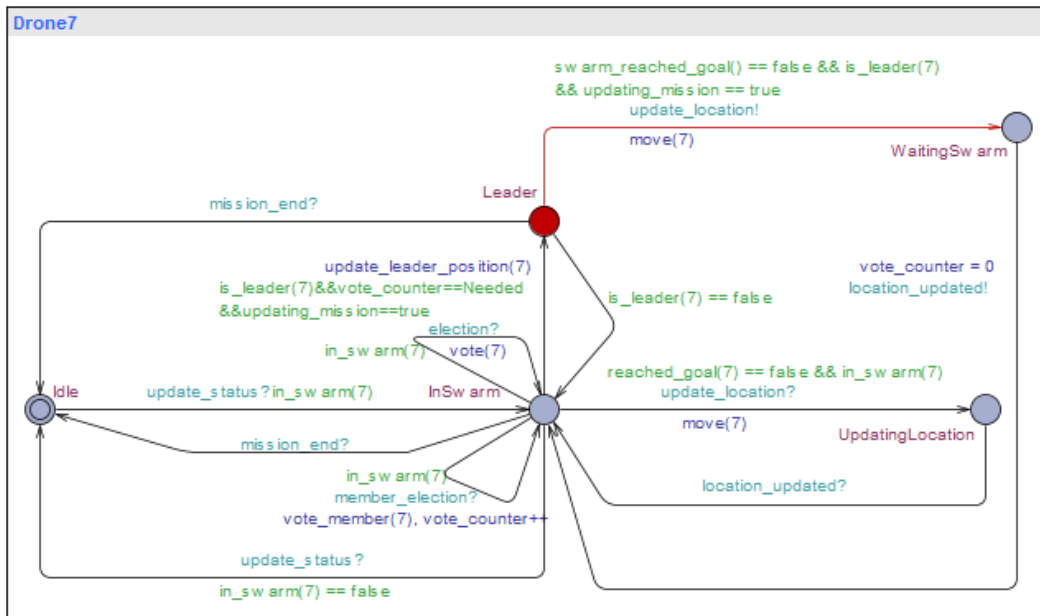


Figure 6.15: Drone 7 Leader State

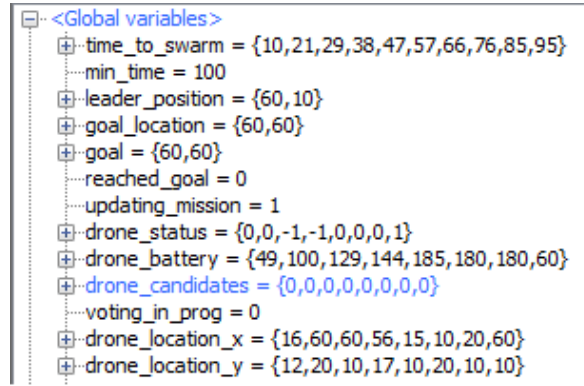


Figure 6.16: Drones at Position 60 in x

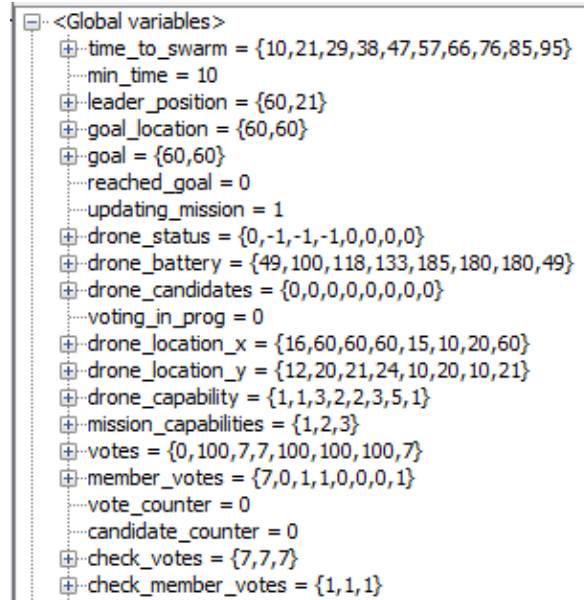


Figure 6.17: Drone 7 Dropped from Swarm and Drone 1 Joins

The drones then continue the mission in the same configuration until they arrive to the goal location. Figure 6.19 shows that drone 3 is the first one to arrive to the end location. Notably, this is not the leader drone. This situation arises because the last drone to join the swarm, which became leader, joined from a position behind the rest.

Finally, Drone 3 waits for the other two drones to arrive at the goal location which culminates the mission. The simulation outcome is shown in Figure 6.20. The same

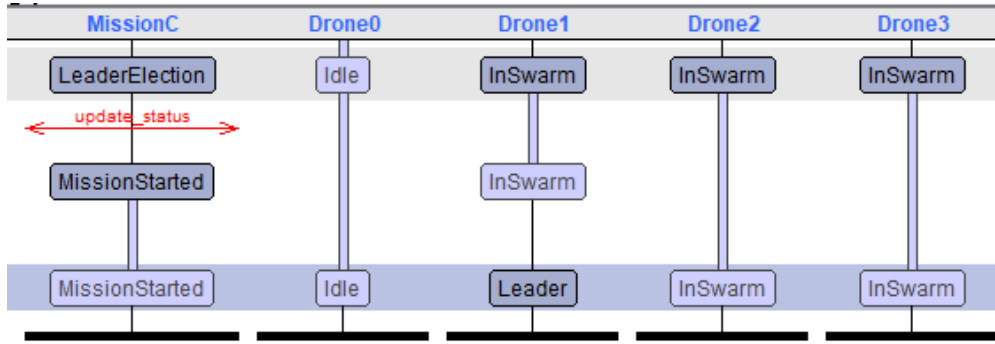


Figure 6.18: Drone1 Becomes Leader

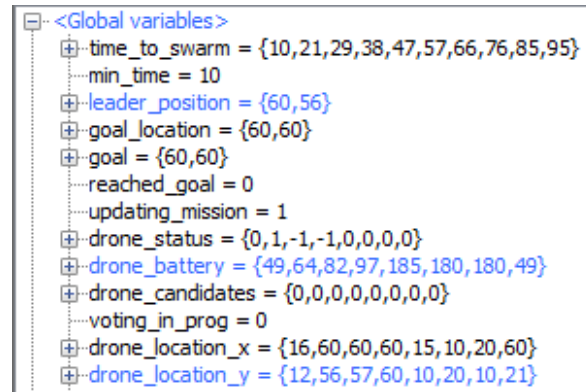


Figure 6.19: Drone 3 Arrives at Goal First

scenario for UPPAAL is shown in Figure 6.21 which reflects the the state of the mission controller is MissionAccomplished. At the same time, all drones transition to the Idle state.

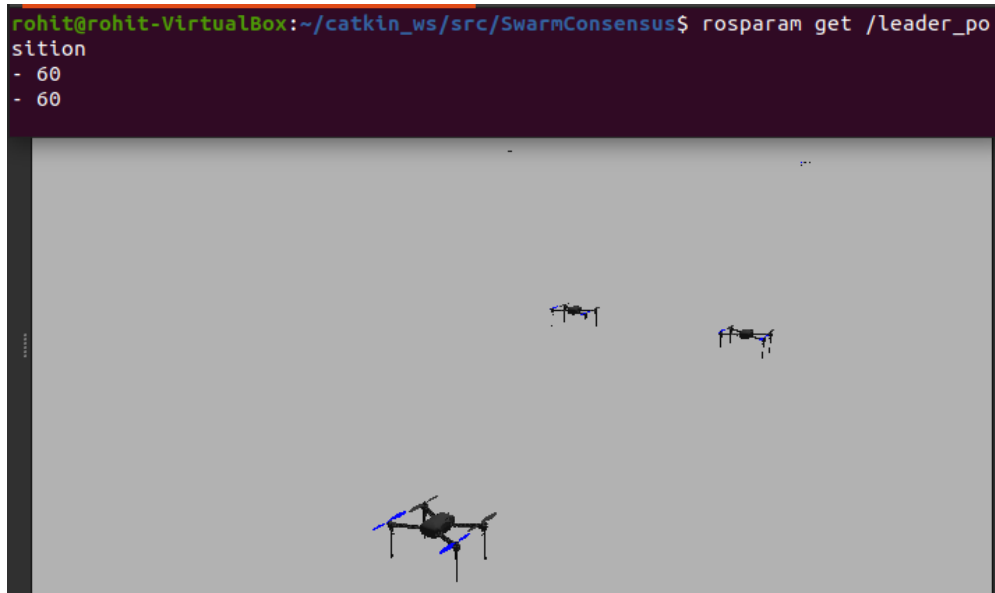


Figure 6.20: ROS Drones at Goal

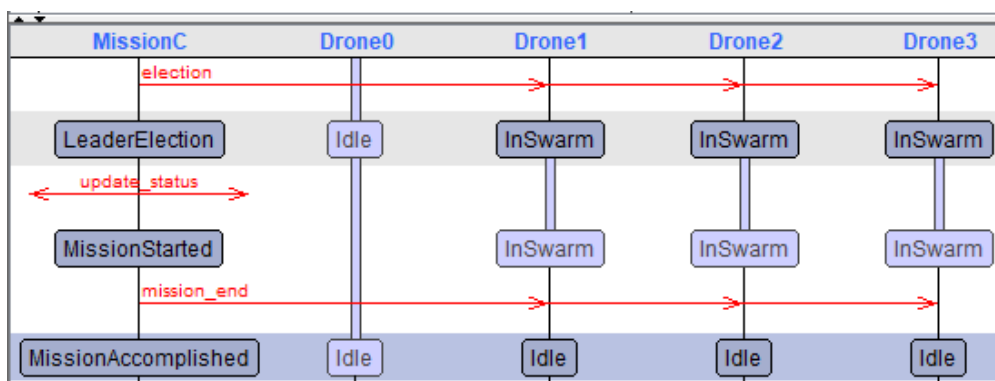


Figure 6.21: Mission Accomplished

Chapter 7

Conclusion and Future Work

The following sections summarize the outcomes of the presented work addressing the limitations and improvements possible for future work.

7.1 Conclusion

In conclusion, the work presented in this thesis showcases a framework to guide the implementation of a verified UPPAAL model in ROS. The framework would be applicable to systems that represent robotics applications where each UPPAAL template represents the behaviour of a certain robot or controller.

Chapter 1 provided background information regarding UAV swarms and their potential applications. It also outlined the motivation behind conducting model checking prior to implementation and its importance in the certification of upcoming industries like UAM. The Chapter ends with a brief statement of the problem at hand and the main software suites used in this work.

Chapter 2 represents the background research conducted to make this work possible. It includes other research efforts where models were transitioned to implementation

through various techniques. Particularly, the work presented in [34] was used as a starting point and the software developed was implemented to automatically translate part of the UPPAAL model to ROS.

Chapter 3 gives a high level overview of the research methodology employed to achieve the desired framework. The problem statement is elaborated to include the case study or target model which is the UAV swarm. The assumptions used in modeling the UPPAAL system are also provided. The chapter culminates with a summary of the translation process and its main components.

Chapter 4 provides the technical details relevant to the framework. Firstly, the Matlab script used to address the abstraction of the drone flight dynamics is explained with its incorporation to the UPPAAL model. The basic UPPAAL concepts needed to understand the Swarm model are presented before addressing each template's behaviour specifically. The most important functions that describe the consensus mechanisms of the swarm model are explained in detail. The chapter ends with an explanation of the system verification process achieved using the Verifier interface in UPPAAL. The properties that prove the correctness of the system are explained to show the robustness of the model.

Chapter 5 shifts focus to the ROS end of the framework. It introduces the basic concepts and structures of ROS which are needed to understand the subsequent implementation. Next, the sections explain the automated translation process with examples from the generated ROS Parameter Server and a function from the Base Class. The high level mapping of the UPPAAL constructs to ROS entities is summarized in tabular format. The chapter is closed with the most substantial contributions in this work which is the mapping of the timed automaton logic to ROS node files. The translation is represented in tabular and algorithmic formats which will be useful for the automation of the process in the future.

Chapter 6 covers the validation of the UPPAAL model and its translation to ROS by observing their behaviour in simulation. The UPPAAL system is first simulated within the tool's symbolic simulator which shows system states step by step. On the other hand, the ROS implementation outlined earlier is tested in the Gazebo 3D simulator which uses a virtualized flight computer provided by the Ardupilot software.

Overall, the framework presented is useful to programmers who intend to design and implement robotics software. The mapping and algorithm provided to port an UPPAAL model to ROS reduces the burden of implementing a verified system. This could encourage developers to invest the time in modeling and verifying a system before the implementation phase. The benefits would include a reduction in error injection during early design phases, the assurance of correct system behaviour, and a reduction in time spent fixing bugs in the implementation phase. Considering the software engineering approach and software development life cycle, the use of modeling to design systems also increases its interpretability. This can be crucial in early stages of development where the development group can discuss design decisions with customers or business teams. It is simpler to explain software design through the graphical representation in model checking than lines of textual code.

7.2 Limitations

The main limitation of the framework presented through this work is the manual processes that tie it together. For example, the array retrieved from the Matlab script representing drone dynamics is manually introduced to the UPPAAL model as a global variable before system execution. Another limitation is that ROS provides a variety of parameters regarding vehicle dynamics and states in real time. These would be more suitable to make decisions during system execution over parameters calculated

in external software. This means that a higher level of abstraction in the modeling process was possible. Instead of populating global arrays with data in the modeling phase, a simple flag variable could be used to trigger actions in ROS. The flag name could be designed to match the name of the ROS topic or parameter that describes the data needed for comparisons or thresholds. This would also make it easier to automate the manual part of the translation.

Limitations specific to the model developed for this work involve the path that the drones take to reach the goal location. As described in the main body of the text, the drones move in rectangular fashion aligning with the target on one axis first and then approaching it. This does not represent the fastest route, which would be the more logical one in a situation like a wildfire mission. The reason behind this choice is that the UPPAAL simulator interface does not support arrays of type double or float.

Regarding the simulation environment the main limitation was the use of a Virtual Machine. The simulation involved running Ardupilot, ROS, and Gazebo simultaneously. The computational burden was heightened by the fact that the swarm involved 8 drones. All of these processes running at a time caused the Real Time Factor of the simulation to be negatively impacted.

Finally, one of the limitations of the mapping of UPPAAL constructs in ROS is the heavy dependence on the ROS Parameter Server. The UPPAAL system relies on using global variables to represent data for each drone while each process has access to read and write. The most direct representation of these mechanics in ROS is the Parameter Server. However, it is not designed for high-performance and its recommended use includes static variables that represent system configurations.

7.3 Future Work

One of the improvements left for future efforts is improving the ROS simulation in Gazebo to make the system more efficient. There is room for improvement in process synchronization which could result in faster transitions and mission completion. The main plan for future work involves fully generalizing and automating the framework presented. The automatic translation from UPPAAL to ROS could be expanded to include the logic in the timed automata. Accomplishing such a feat would mean that most robotic systems modeled in UPPAAL could be automatically implemented in ROS for SITL testing and simulation. This would represent significant cuts in time of development for robotics applications. Furthermore, it would help in the process of applying the developed software to real world hardware because of the hardware virtualization aspect of SITL testing.

Bibliography

- [1] Ros with mavros installation guide. *PX4 User Guide*, Jul 2022.
- [2] Jaleel H. Abdelkader M., Güler S. and Shamma J. Aerial swarms: Recent applications and challenges. *Current Robotics Report*, (2):309 – 320, 2021.
- [3] ArduPilot. Ardupilot. *ArduPilot.org*, Nov 2022.
- [4] Matt Asay. Building the future of robots development with ros 2. *Amazon Web Services*, Nov 2020.
- [5] Justin Bachman. Embraer’s air mobility unit agrees to \$2.4 billion spac deal. *Bloomberg*, 2021.
- [6] Christel Baier and Joost-Pieter Katoen. *Principles of model checking. [electronic resource]*. MIT Press, 2008.
- [7] S. Baldi, D. Sun, X. Xia, G. Zhou, and D. Liu. Ardupilot-based adaptive autopilot: Architecture and software-in-the-loop experiments. *IEEE Transactions on Aerospace and Electronic Systems, Aerospace and Electronic Systems, IEEE Transactions on, IEEE Trans. Aerosp. Electron. Syst*, 58(5):4473 – 4485, 2022.
- [8] BBC. Nhs trials using drones to deliver chemotherapy drugs. *BBC News*, July 2022.

- [9] Simone Cirani, Marco Picone, Gianluigi Ferrari, and Luca Veltri. *Internet of things : architectures, protocols and standards. [electronic resource]*. Wiley, 2019.
- [10] Amanda Dattalo. Ros introduction. *ros.org*, Aug 2018.
- [11] Saeid Dehnavi, Ali Sedaghatbaf, Bahare Salmani, Marjan Sirjani, Mehdi Kargahi, and Ehsan Khamespanah. Rerebeca : A new framework to design verified ros-based robotic programs. 08 2019.
- [12] The Standish Group. Chaos 2020: Beyond infinity, 2020.
- [13] Anubhav Gupta, Siddhartha Bhattacharyya, and S. Vadivel. Can model checking assure, distributed autonomous systems agree? an urban air mobility case study. *International Journal of Advanced Computer Science and Applications*, 11, 01 2020.
- [14] R. Halder, J. Proenca, N. Macedo, and A. Santos. Formal verification of ros-based robotic applications using timed-automata. *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE), Formal Methods in Software Engineering (FormaliSE), 2017 IEEE/ACM 5th International FME Workshop on, FORMALISE*, pages 44 – 50, 2017.
- [15] David Hambling. What are drone swarms and why does every military suddenly want one? *Forbes*, 2021.
- [16] B. Ingersoll, K. Ingersoll, P. Defranco, and A. Ning. Uav path-planning using bézier curves and a receding horizon approach. In *AIAA Modeling and Simulation Technologies Conference, 2016*, number AIAA Modeling and Simulation Technologies Conference, page 14p., Department of Mechanical Engineering, Brigham Young University, 2016.

- [17] Bryce Ingersoll. uav-path-optimization. <https://github.com/byuflowlab/uav-path-optimization>, January 2016.
- [18] Capers Jones. *Applied software measurement : global analysis of productivity and quality*. McGraw-Hill's AccessEngineering. McGraw-Hill, 2008.
- [19] Lentin Joseph. *ROS Robotics Projects*. Packt Publishing, 2017.
- [20] Cedric Justin, Alexia Payan, Simon Briceno, and Dimitri Mavris. Operational and economic feasibility of electric thin haul transportation. 06 2017.
- [21] Professor Paul Pettersson Kim G. Larsen and Professor Wang Yi. UPPAAL introduction. <https://uppaal.org/>. Accessed: 2022-04-02.
- [22] Savas Konur, Clare Dixon, and Michael Fisher. Analysing robot swarm behaviour via probabilistic model checking. *Robotics and Autonomous Systems*, 60(2):199–213, 2012.
- [23] Daniil Kozlov. Comparison of reinforcement learning algorithms for motion control of an autonomous robot in gazebo simulator. *2021 International Conference on Information Technology and Nanotechnology (ITNT), Information Technology and Nanotechnology (ITNT), 2021 International Conference on*, pages 1 – 5, 2021.
- [24] Herb Krasner. The cost of poor software quality in the us: A 2020 report. 2021.
- [25] Matheus Ladeira, Yassine Ouhammou, and Emmanuel Grolleau. Robmex: Ros-based modelling framework for end-users and experts. *Journal of Systems Architecture*, 117, 2021.

- [26] Xinxin Li, Rui Wang, Yu Jiang, Yong Guan, Xiaojuan Li, and Xiaoyu Song. Formal modeling and automatic code synthesis for robot system. *2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS), Engineering of Complex Computer Systems (ICECCS), 2017 22nd International Conference on*, pages 146 – 149, 2017.
- [27] S. Liu and J. Gaudiot. Rise of the autonomous machines. *Computer*, 55(1):64 – 73, 2022.
- [28] Wenrui Meng, Junkil Park, Oleg Sokolsky, Stephanie Weirich, and Insup Lee. *Verified ROS-Based Deployment of Platform-Independent Control Systems.*, volume 9058 of *Lecture Notes in Computer Science. 9058*. Springer International Publishing, 2015.
- [29] Sangwoo Moon, John J. Bird, Steve Borenstein, and Eric W. Frew. A gazebo/ros-based communication-realistic simulator for networked suas. In *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1819–1827, 2020.
- [30] Majda Moussa and Giovanni Beltrame. On the robustness of consensus-based behaviors for robot swarms. *Swarm Intelligence*, 14(3):205 – 231, 2020.
- [31] Globe Newswire. United States Urban Air Mobility (UAM) Market Report 2021: Market is Estimated to Reach \$18.81 Billion in 2035, at a CAGR of 23.12% during 2023-2035. Accessed: 2022-02-07.
- [32] Annie Palmer. Amazon wins faa approval for prime air drone delivery fleet. *CNBC*, 2020.
- [33] Adithya T. Praveen. uppaal2ros. <https://github.com/adithya-tp/uppaal2ros>, January 2021.

- [34] Adithya T. Praveen, Anubhav Gupta, Siddhartha Bhattacharyya, and Raja Muthalagu. Assuring behavior of multi-robot autonomous systems with translation from formal verification to ros simulation. *IEEE Systems Journal*, pages 1–9, 2022.
- [35] Umer Salman. Drone software development tutorials. https://www.youtube.com/playlist?list=PLy9nLDKxDN683GqAiJ4IVLquYBod_2oA6.
- [36] Umer Salman. Intelligent-Quads. <https://github.com/Intelligent-Quads>, 2019.
- [37] Fiona Saunders, Andrew Gale, and Andrew Sherry. Understanding project uncertainty in safety-critical industries. 04 2013.
- [38] Amazon Staff. Amazon prime air prepares for drone deliveries. *US About Amazon*, June 2022.
- [39] Sidra Sultana and Fahim Arif. From verification to implementation: Uppaal to c++. 2016.
- [40] Mohamed Fasil Syed Ahamed, Girma Tewolde, and Jaerock Kwon. Software-in-the-loop modeling and simulation framework for autonomous vehicles. *2018 IEEE International Conference on Electro/Information Technology (EIT), Electro/Information Technology (EIT), 2018 IEEE International Conference on*, pages 0305 – 0310, 2018.
- [41] S. Tangirala, R. Kumar, S. Bhattacharyya, M. O’Connor, and L.E. Holloway. Hybrid-model based hierarchical mission control architecture for autonomous underwater vehicles. *Proceedings of the 2005, American Control Conference, 2005., American Control Conference, 2005. Proceedings of the 2005, American Control Conference*, page 668, 2005.

- [42] Abenezer G. Taye, Josh Bertram, Chuchu Fan, and Peng Wei. *Reachability based Online Safety Verification for High-Density Urban Air Mobility Trajectory Planning*.
- [43] Reno University of Nevada. What are intelligent systems? <https://www.unr.edu/cse/undergraduates/prospective-students/what-are-intelligent-systems>. Accessed: 2022-02-07.
- [44] Qinshuang Wei, Gustav Nilsson, and Samuel Coogan. Safety verification for urban air mobility scheduling. *IFAC-PapersOnLine*, 55(13):306–311, 2022. 9th IFAC Conference on Networked Systems NECSYS 2022.
- [45] Hwee Yan. ardupilot-gazebo-ros-guide. <https://github.com/yanhwee/ardupilot-gazebo-ros-guide>, March 2020.
- [46] Xin Zhou, Xiangyong Wen, Zhepei Wang, Yuman Gao, Haojia Li, Qianhao Wang, Tiankai Yang, Haojian Lu, Yanjun Cao, Chao Xu, and Fei Gao. Swarm of micro flying robots in the wild. *Science Robotics*, 7(66), 2022.

Appendix A

Drone Dynamics Matlab Script

The main script of the Matlab program developed by Ingersoll et al. and published in [16] is included below.

```
1 % ————— Main File ————— %
2 % Author : Bryce Ingersoll
3 % Institution: Brigham Young University , FLOW Lab
4 % Last Revised : 7/6/16
5 % ————— %
6
7 %clear; clc; close all;
8
9 %Add paths
10 addpath(genpath( '.\Objective_Functions\' ));
11 addpath(genpath( '.\Constraints\' ));
12 addpath(genpath( '.\ColorPath\' ));
13 addpath(genpath( '.\Compare\' ));
```

```

14 addpath(genpath('.\OptimalPathGuesses\'));
15 addpath(genpath('.\CalculateEnergyUse\'));
16
17 %profiling tools
18 %profile on
19
20 %—————global variables—————%
21 global xf; %final position
22 global x0; %current starting pointPath_bez
23 global step_max; %max step distance
24 global step_min; %minimum step distance
25 global t; %parameterization variable
26 global n_obs; %number of obstacles
27 global obs; %positions of obstacles
28 global obs_rad; %radius of obstacles
29 global turn_r; %minimum turn radius
30 global Pmid; %needed to match derivatives
31 global num_path; %number of segments optimized
32 global x_new;
33 global Dynamic_Obstacles;
34 global x_next; %used in multi_start function
35 global uav_ws; %UAV wing span
36 global start;
37 global initial; % to calculate d_l_min
38 initial = 1;
39 global uav_finite_size;

```



```

40 global rho f W span eo;
41 global summer_c cool_c copper_c parula_c winter_c blue_red
    blue_magenta_red green_purple blue_gray_red
    shortened_viridis_c shortened_inferno_c;
42 global shortened_parula_c;
43 global obj_grad cons_grad ag acg;
44 global max_speed min_speed D_eta_opt;
45 global l1_l_last;
46
47 %-----Algorithm Options-----%
48
49 % use genetic algorithm
50 use_ga = 0;
51
52 Dynamic_Obstacles = 0;
53
54 num_path = 3; %Receding Horizon Approach (any
    number really , but 3 is standard)
55 ms_i = 3; %number of guesses for multi start
    (up to 8 for now, up to 3 for smart)
56 uav_finite_size = 1; %input whether want to include UAV
    size
57 check_viability = 1; %Exits if unable to find viable
    path
58
59 %Objective Function

```

```

60 optimize_energy_use = 0;    %changes which objective function
    is used
61 optimize_time = 1;        %if both are zero , then path length is
    optimized
62
63 max_func_evals = 10000;
64 max_iter = 50000;
65
66 % Plot Options
67 totl = 1;                %turn off tick labels
68 square_axes = 1;        %Square Axes
69 radar = 0;              %Plots UAV's limit of sight
70 show_sp = 0;            %Plots P1 of Bezier curve
71
72 Show_Steps = 0;         %Needs to be turned on when
    Dynamic_Obstacles is turned on
73 linewidth = 4;          %Line width of traversed path segment
74 traversedwidth = 2;
75 dashedwidth = 2;
76
77 fwidth = 2;             %width of UAV path in FinalPlot.m
78 show_end = 0;           %for calc_fig
79 compare_num_path = 0;
80 save_path = 0;          %save path data to use in compare
81 sds = 0;                %Allows a closer view of dynamic
    obstacle avoidance

```

```

82 cx = 50;
83
84 %plot color options
85 speed_color = 1;           %use if you want color to represent
    speed
86 d_speed_color = 0;         %use if you want color to be
    discretized over path length
87 cb = 1;                    %color brightness
88
89 summer_c = 0;              % http://www.mathworks.com/help/
    matlab/ref/colormap.html#buq1hym
90 cool_c = 0;
91 copper_c = 0;
92 parula_c = 0;
93 winter_c = 0;
94 blue_red = 0;
95 blue_magenta_red = 0;
96 green_purple = 0;
97 blue_gray_red = 0;
98 shortened_parula_c = 1;
99 shortened_viridis_c = 0;
100 shortened_inferno_c = 0;
101 color_bar = 1;
102 %-----%
103

```

```

104 create_video = 1;           %saves the solutions of the
    multistart approach at each iteration
105
106 % Gradient Calculation Options
107 obj_grad = 1;               %if this is 1 and below line is 0,
    complex step method will be used to calculate gradients
108 analytic_gradients = 1;
109 ag = analytic_gradients;
110
111 cons_grad = 1;              %if this is 1 and below line is 0,
    complex step method will be used to calculate gradients
112 analytic_constraint_gradients = 1;
113 acg = analytic_constraint_gradients;
114
115 %—————plane geometry/info—————%
116 %UAV parameter values
117 rho = 1.225; %air density
118 f = .2; %equivalent parasite area
119 W = 10; %weight of aircraft
120 span = .20; %span
121 eo = 0.9; %Oswald's efficiency factor
122
123 if optimize_energy_use == 1
124     %Defined in paper (2nd column, page 2)
125     A = rho*f/(2*W);
126     B = 2*W/(rho*span^2*pi*eo);

```

```

127
128     %find minimum d_l, and minimum efficiency
129     if initial == 1
130         V_possible = 0.1 : 0.01 : 25;
131         !
132         for i = 1 : length(V_possible)
133
134             D_L = A*V_possible(i)^2 + B/V_possible(i)^2; % we
                    want to maximize l_d, or minimize d_l
135
136             eta_pos = calc_eff(V_possible(i));
137
138             %calculate D_L/eta
139             D_eta(i) = D_L/eta_pos;
140         end
141
142         %find optimal D_eta
143         D_eta_opt = min(D_eta);
144
145     end
146 end
147 % ----- %
148
149 l = 0;
150
151 %parameterization vector t

```

```

152 global delta_t;
153 t = linspace(0,1,11);
154 delta_t = t(2) - t(1);
155
156
157 %for plot_both function
158 %global Path_bez;
159
160 turn_r = 5; %turn radius, m
161
162 %maximum/stall speed, m/s
163 max_speed = 15.0;
164 min_speed = 10.0;
165
166 l_l_last = (min_speed)/(length(t));
167
168 %transalte UAV information to fit with algorithm
169 step_max = max_speed; %/2;
170 step_min = min_speed; %/2;
171
172 %Wing span of UAV
173 if uav_finite_size == 1
174     uav_ws = 1.0; %UAV wing span
175 else
176     uav_ws = 0.001;
177 end

```

```

178
179 %starting/ending position of plane
180 x_sp = [0,0];
181 x0 = x_sp;
182 %xf = [45,45];
183 Bez_points = [];
184 lr = 8; %landing zone radius; should be  $\leq 15$ 
185 %-----%
186
187 %-----static obstacle information-----%
188 %rng(3); %50/4/3
189 %rng(4); %49/4/3
190 %rng(59); %54/4/3 – use for sds, ms_i = 3, 34/4/3
191 %rng(60); %50/4/3
192 %rng(13); %40/4/3
193 %rng(15); %40/4/3
194 %rng(20); %40/4/3
195 %rng(8);
196 rng(8);
197
198 %n_obs = 5; %number of static obstacles
199 n_obs = xf(1)/10; %number of static obstacles
200 obs = rand(n_obs,2)*xf(1)+5; %obstacle locations
201
202
203 rng(5); %for partially random obstacle size

```

```

204 obs_rad = (15-uav_ws) + rand(n_obs,1)*3; %obstacle radius
205 %-----%
206
207 % calculate density
208 obs_density = calc_obs_den(n_obs, obs, obs_rad, uav_ws);
209
210 %-----dynamic obstacle information-----%
211 if Dynamic_Obstacles == 1
212
213     global n_obsd obs_d_sp obs_d_v obs_d_s obs_d_cp;
214
215     %choose 1-4 for cases (see function for description)
216     [n_obsd, obs_d_sp, obs_d_s, obs_d_v] = dyn_case(5);
217
218     obs_d_s = obs_d_s-ones(n_obsd,1)*uav_ws; %size of
           obstacles, also used (5)
219     obs_d_cp = obs_d_sp; %current position of obstacles
220     obs_d_cp_hist(1,:,1) = obs_d_sp(1,:);
221 end
222 %-----%
223
224 % for make_video
225 if create_video == 1
226
227     solution1 = [];
228     solution2 = [];

```



```

229     solution3 = [];
230     solution4 = [];
231     solution5 = [];
232
233 end
234
235
236 %----- optimizer ----- fmincon
      -----%
237 %unused parts in fmincon
238 A = [];
239 b = [];
240 Aeq = [];
241 beq = [];
242 %lb = -10*ones(2*num_path,2);
243 %ub = 110*ones(2*num_path,2);
244 lb = [];
245 ub = [];
246
247 %Pmed initialization , used to match up derivatives between
      paths
248 Pmid = [-min_speed/2,-min_speed/2];
249 %Pmid = [-3,-3];
250
251 Path_bez = [];
252

```

```

253 path_start = [];
254
255 %initial guess(es)
256 start = 0;
257
258 xi = multi_start(ms_i);
259
260 %start
261 start = 1;
262
263 %x_new is not close to final position
264 x_new = zeros(2*num_path,2);
265 x_next = x_new;
266
267 % note: each iteration of while loop represents some time step
        , in which
268 % UAV travels on path and dynamic obstacles move
269
270 %fmincon options
271 if obj_grad == 1 && cons_grad == 1
272     options = optimoptions('fmincon','Algorithm','sqp','
        MaxFunEvals',max_func_evals,'MaxIter',max_iter,...
273     'GradObj','on','GradCon','on','DerivativeCheck','off')
        ;
274 elseif obj_grad == 0 && cons_grad == 1

```

```

275     options = optimoptions('fmincon','Algorithm','sqp',
        MaxFunEvals,max_func_evals,'MaxIter',max_iter,...
276     'GradObj','off','GradCon','on','DerivativeCheck','off'
        );
277 elseif obj_grad == 1 && cons_grad == 0
278     options = optimoptions('fmincon','Algorithm','sqp',
        MaxFunEvals,max_func_evals,'MaxIter',max_iter,...
279     'GradObj','on','GradCon','off','DerivativeCheck','off'
        );
280 else
281     options = optimoptions('fmincon','Algorithm','sqp',
        MaxFunEvals,max_func_evals,'MaxIter',max_iter,...
282     'GradObj','off','GradCon','off');
283 end
284
285 tic % begin optimization time
286
287 while ( ( (x_next(2*num_path,1)-xf(1))^2 + (x_next(2*num_path
        ,2)-xf(2))^2 )^0.5 > lr )
288
289     %record number of paths
290     l = l + 1;
291
292
293     for i = 1 : ms_i %multistart approach to find best
        solution

```

```

294
295 %choose objective function
296 if optimize_energy_use == 1
297
298     if use_ga == 1
299         error('Not ready yet for ga');
300     else
301         [x_new(:, :, i), ~, e(i, l)] = fmincon(@opt_e, xi
302             (:, :, i), A, b, Aeq, beq, lb, ub, @cons,
303             options);
304     end
305
306 elseif optimize_time == 1
307
308     if use_ga == 1
309         error('Not ready yet for ga');
310     else
311         [x_new(:, :, i), ~, e(i, l)] = fmincon(@opt_t, xi
312             (:, :, i), A, b, Aeq, beq, lb, ub, @cons,
313             options);
314     end
315
316 else
317
318     if use_ga == 1
319         error('Not ready yet for ga');

```

```

316         else
317             [x_new(:, :, i), ~, e(i, 1)] = fmincon(@opt_d, xi
                (:, :, i), A, b, Aeq, beq, lb, ub, @cons,
                options);
318         end
319
320     end
321
322     %         %check curvature
323     %         c = check_curvature_new(i);
324     %
325     %         %if constraints are violated, make
326     %         infeasible
327     %         if any(c > 0)
328     %             e(i, 1) = -2;
329     %         end
330
331     end
332
333     for i = 1 : ms_i %calculate how good solutions are
334
335         % For make_video
336         if create_video == 1
337
338             if i == 1
339                 solution1 = [solution1; x_new(:, :, i)];
340             elseif i == 2

```

```

339         solution2 = [solution2; x_new(:, :, i)];
340     elseif i == 3
341         solution3 = [solution3; x_new(:, :, i)];
342     elseif i == 4
343         solution4 = [solution4; x_new(:, :, i)];
344     elseif i == 5
345         solution5 = [solution5; x_new(:, :, i)];
346     end
347 end
348
349 if optimize_energy_use == 1
350     d_check(i) = opt_e(x_new(:, :, i));
351
352 elseif optimize_time == 1
353     d_check(i) = opt_t(x_new(:, :, i));
354
355 else
356     d_check(i) = opt_d(x_new(:, :, i));
357
358 end
359
360 %'remove' solutions that converged to an infeasible
    point
361
362 if e(i, 1) == -2
363

```

```

364         d_check(i) = d_check(i)*10;
365
366     end
367
368
369 end
370
371 %Check for viable paths
372 check = (e == -2);
373 if all(check(:,1)) == 1 && check_viability == 1
374     %error('Unable to find viable path. ');
375 end
376
377 for i = 1 : ms_i %choose best solution , use for next part
378
379     if d_check(i) == min(d_check)
380
381         x_next = x_new(:, :, i);
382
383     end
384 end
385
386 %
387 initial = 0;
388
389 % makes the path of the UAV for this section

```

```

390     for i = 1 : length(t)
391
392         path_part(i,:) = (1-t(i))^2*x0(1,:) + 2*(1-t(i))*t(i)*
            x_next(1,:)+t(i)^2*x_next(2,:);
393
394         %         if i > 1
395         %         norm(path_part(i,:)-path_part(i-1,:))
396         %         end
397
398     end
399
400     %make the planned path of the UAV
401     if num_path > 1
402         for j = 1 : (num_path-1)
403             for i = 1 : length(t)
404                 path_planned(i+(j-1)*length(t),:) = (1-t(i))
                    ^2*x_next(2*j,:) + 2*(1-t(i))*t(i)*x_next
                    (2*j+1,:)+t(i)^2*x_next(2*j+2,:);
405             end
406         end
407     end
408
409     if Show_Steps == 1
410
411         plot_int_steps(1, square_axes, color_bar, totl, x_sp,
            cx, speed_color, path_part, path_planned, Path_bez,

```



```

        d_speed_color , cb...
412         ,linewidth , traversedwidth , dashedwidth , radar ,
            show_sp , show_end , sds , lr );
413     end
414
415     %
    _____%
416
417     %record where start of each path is
418     path_start = [path_start; path_part(1,:)];
419
420     %continues the path which will be plotted
421     Path_bez = [Path_bez; path_part];
422
423     l_l_last = norm(Path_bez(length(Path_bez),:)-Path_bez(
        length(Path_bez)-1,:));
424
425     %set new starting point
426     x0 = x_next(2,:);
427
428     %set Pmid
429     Pmid = x_next(1,:);
430
431     %choose new guess for next iteration
432     xi = multi_start(ms_i);

```

```

433
434     %print current location
435     x_next(2,:)
436
437     Bez_points = [Bez_points; x_next(1:2,:)];
438
439 end %while
440
441 toc % end optimization time
442
443 Bez_points = [Bez_points; x_next(3:num_path*2,:)];
444
445 % Final Plot
446 FinalPlot(path_start, Path_bez, l, square_axes, totl,
            color_bar, speed_color...
447 , delta_t, d_speed_color, cb, cx, lr, x_sp, fwidth);
448
449 %compare paths created using various number of look ahead
    paths
450 if compare_num_path == 1
451
452     if num_path == 1
453         save( '.\Compare\path_1.txt', 'Path_bez', '-ascii' );
454         save( '.\Compare\start_1.txt', 'path_start', '-ascii' );
455     elseif num_path == 2
456         save( '.\Compare\path_2.txt', 'Path_bez', '-ascii' );

```

```

457         save( '.\Compare\start_2.txt', 'path_start', '-ascii' );
458     elseif num_path == 3
459         save( '.\Compare\path_3.txt', 'Path_bez', '-ascii' );
460         save( '.\Compare\start_3.txt', 'path_start', '-ascii' );
461     elseif num_path == 4
462         save( '.\Compare\path_4.txt', 'Path_bez', '-ascii' );
463         save( '.\Compare\start_4.txt', 'path_start', '-ascii' );
464     elseif num_path == 5
465         save( '.\Compare\path_5.txt', 'Path_bez', '-ascii' );
466         save( '.\Compare\start_5.txt', 'path_start', '-ascii' );
467     elseif num_path == 6
468         save( '.\Compare\path_6.txt', 'Path_bez', '-ascii' );
469         save( '.\Compare\start_6.txt', 'path_start', '-ascii' );
470     end
471
472 end
473
474 %add planned path
475 Path_bez = [Path_bez; path_planned(2:length(path_planned),:)] ;
476
477 %save path info to use in 'compare file'
478 if save_path == 1
479
480     if optimize_energy_use == 1
481
482         save( '.\Compare\path_e.txt', 'Path_bez', '-ascii' );

```

```

483         save( '.\Compare\start_e.txt', 'path_start', '-ascii' );
484
485     elseif optimize_time == 1
486
487         save( '.\Compare\path_t.txt', 'Path_bez', '-ascii' );
488         save( '.\Compare\start_t.txt', 'path_start', '-ascii' );
489
490     else
491         save( '.\Compare\path_d.txt', 'Path_bez', '-ascii' );
492         save( '.\Compare\start_d.txt', 'path_start', '-ascii' );
493
494     end
495 end
496
497
498 %output of compare (distance, time, energy)
499 [td, tt, te] = compare_of(Path_bez, Bez_points,
        optimize_energy_use, optimize_time);
500
501 %profiling tools
502 %profiling_info = profile('info');
503
504 %toc % end optimization and plotting time

```

Appendix B

Uppaal Drone Swarm System XML

The XML file representing the Uppaal system developed for this work is included below.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!DOCTYPE nta PUBLIC "-//Uppaal Team//DTD Flat System 1.1//EN" 'http://
   www.it.uu.se/research/group/darts/uppaal/flat-1.2.dtd'>
3 <nta>
4   <declaration> // Place global declarations here.
5     // Assume the communication reach is 10: the maximum
6     // reach of control tower and reach of leader drone
7
8     const int comm_reach = 10;
9
10    int time_to_swarm[comm_reach] = {10, 21, 29, 38, 47, 57, 66, 76, 85, 95};
11
12    int min_time = 0;
13
```

```

14 // Total # of drones
15
16 const int N = 8;
17
18 // Drones in mission
19 const int Needed = 3;
20
21 //leader position, Initial leader is the tower
22 int leader_position[2] = {0,0};
23
24 // define global goal
25 int goal_location[2];
26
27 // fire location / swarm goal
28 int goal[2] = {60,60};
29
30 // variable that maps to function swarm_reached_goal
31 bool reached_goal = 0;
32
33 // Mission update control
34 bool updating_mission = 1;
35
36 //Each drone ID is represented by the array index.
37 // drone characterization: 0 = not in swarm/network, -1 = loyal wingman, 1 =
    leader // consider template for each

```

```

38 // this should allow for easy verification --> not more than one leader, at
    least one leader, etc
39 // also useful to determine availability check, if it is already in swarm it cannot
    join, if it is not a loyal wingman it cant be elected leader, etc
40 // initialization leader? any loyal wingman? or none.
41 int drone_status[N] = {-1,0,-1,-1,0,0,0,0};
42
43 // List of drone batteries — currently 3 switches in leader
44 int drone_battery[N] = {55, 100, 175, 190, 185, 180, 180, 100};
45
46 // List of drone candidates for swarm
47 // if 1, then that drone number (index) is a candidate to join swarm
48 int drone_candidates[N] = {0, 0, 0, 0, 0, 0, 0, 0};
49
50 // variable indicating if voting is in progress
51 bool voting_in_prog = 0;
52
53 // List of drone locations
54 int drone_location_x[N] = {10,60,14,10,15,10,20,20};
55 int drone_location_y[N] = {12,20,10,17,10,20,10,10};
56
57 // TODO: Potentially put drone data into structs
58
59 // variable to identify capabilities of each drone
60 // use case: wild land firefighting / communication towers burned down
61 // 0 = surveillance / good camera

```

```

62 // 1 = communications / long range external entity
63 // 2 = medical supply payload
64 // 3 = water dispenser
65 // 4 = chemical dispenser / fire retardant
66 // store the required drone capabilities in array
67 int drone_capability[N] = {1,1,3,2,2,3,5,1};
68
69 // capabilities needed for this mission
70 int mission_capabilities[Needed] = {1,2,3};
71
72
73
74 // Consensus
75
76 // drones populate this array with their vote for leader
77 // the number represents the drone if for whom each one votes
78 // if all elements match, then we have consensus
79
80 // initialization of votes for each drone
81 int votes[N] = {100,100,100,100,100,100,100,100};
82
83 // votes for members
84 int member_votes[N] = {0, 0, 0, 0, 0, 0, 0, 0};
85
86 // counts how many votes have already been casted during election process
87 int vote_counter = 0;

```



```

88
89     int candidate_counter = 0;
90
91     // stores the votes by the active swarm members
92     int check_votes[Needed];
93
94     // stores the votes by the active swarm members
95     int check_member_votes[Needed];
96
97
98     // voter id
99     int id_voter;
100
101     //the time it would take for a drone to reach swarm
102     int drone_time[N];
103     int dist = 0;
104
105     //election synchronization variable
106     broadcast chan election;
107
108     // member electrion synch
109     broadcast chan member_election;
110
111     // Synchronisation event when the drone starts
112     broadcast chan update_status;
113

```

```

114    // Synchronization event for mission end / mission accomplished
115    broadcast chan mission_end;
116
117    // Synchronization events for movement
118    broadcast chan update_location;
119    broadcast chan location_updated;
120
121
122
123 </declaration>
124 <template>
125     <name>Drone</name>
126     <parameter>int id, int &cap, int &battery</parameter>
127     <declaration>
128 // location coordinates
129 int x;
130 int y;
131
132 // function that checks if a given drone is leader
133 bool is_leader(int id){
134     return drone_status[id] == 1;
135 }
136
137 // Check if drone is in swarm
138 bool in_swarm(int id){
139     return drone_status[id] != 0;

```

```

140     }
141
142
143 // Find the current drone in the swarm with capability return its id
144     int find_drone_in_swarm(int capability){
145         for (id : int[0,N-1])
146         {
147             if(drone_status[id] != 0 && drone_capability[id] == capability)
148                 {
149                     return id;
150                 }
151         }
152     }
153
154
155 // Leader only stops issuing move commands if everybody reached location.
156 // Mission is only complete when swarm reached goal
157     bool swarm_reached_goal(){
158         bool reached = true;
159         for (id : int[0,N-1])
160         {
161             if(in_swarm(id))
162             {
163                 reached = reached && drone_location_x[id] ==
                    goal_location[0] && drone_location_y[id] ==

```

```

        goal_location[1];
164     }
165 }
166
167     return reached;
168 }
169
170
171 // updates the variable that holds a copy of the position of the leader drone
172 void update_leader_position(int id){
173     leader_position[0] = drone_location_x[id];
174     leader_position[1] = drone_location_y[id];
175 }
176
177 // checks if a given drone has reached the goal location
178 bool reached_goal(int id){
179     return drone_location_x[id] == goal_location[0] && drone_location_y[id]
        == goal_location[1];
180 }
181
182 // linear_distance() function returns the calculated linear distance i.e the linear
        distance of the drone from the requesting node
183
184 int linear_distance(int goal[2], int a, int b) {
185     int x_coord = goal[0];
186     int y_coord = goal[1];

```

```

187
188     int xsquare = (x_coord-a)*(x_coord-a);
189     int ysquare = (y_coord-b)*(y_coord-b);
190     int sum1 = xsquare + ysquare;
191     sum1 = fint(sqrt(sum1));
192     return sum1; // fint fuction converts boolean or floating point value to
        integer
193 }
194
195
196 void calculate_drone_times(int cap){
197     for (i : int[0,N-1])
198     {
199         if(drone_candidates[i] == 1 && drone_capability[i] == cap)
200         {
201             dist = linear_distance(leader_position, drone_location_x[i],
                drone_location_y[i]);
202             if(dist > 0 && dist < 9){
203                 drone_time[i] = time_to_swarm[dist-1];
204             }
205         }
206     }
207 }
208
209
210 // Move closer to the goal, if haven't reached it yet

```

```

211 void move(int id){
212     if (drone_location_x[id] != goal_location[0]){
213         drone_location_x[id] = drone_location_x[id] + 1;
214         drone_battery[id] -= 1;
215     }
216     else {
217         if (drone_location_y[id] != goal_location[1])
218             drone_location_y[id] = drone_location_y[id] + 1;
219         drone_battery[id] -= 1;
220     }
221
222     if(is_leader(id)) update_leader_position(id);
223
224     x = drone_location_x[id];
225     y = drone_location_y[id];
226 }
227
228
229
230 // function for drone to cast vote for leader election
231 void vote(int id){
232
233     for (i : int[0,N-1])
234     {
235         if(drone_status[i] == -1 && drone_capability[i] == 1)
236         {

```

```

237         votes[id]=i;
238     }
239
240 }
241     vote_counter++;
242 }
243
244 int find_min(int arr[N], int capability){
245
246     int min = 100;
247
248     for(i:int[0,N-1])
249     {
250         if(drone_capability[i] == capability && drone_candidates[i] == 1)
251         {
252             if(arr[i] != 0 && arr[i]<min)
253             {
254                 min = arr[i];
255             }
256         }
257     }
258     return min;
259 }
260
261
262 // function for drone to cast vote on member election

```

```

263 void vote_member(int id){
264
265
266     for (i : int[0,N-1])
267     {
268         if(drone_candidates[i] == 1)
269         {
270             int current_drone = find_drone_in_swarm(drone_capability[i]); //problem is:
                taking min time from different capability
271             calculate_drone_times(drone_capability[i]);
272             min_time = find_min(drone_time, drone_capability[current_drone]);
273             if(drone_time[i] == min_time)
274             {
275                 if((drone_battery[current_drone]<50) && drone_battery[i]
                    > drone_battery[current_drone])
276                 {
277                     //vote for drone to be part of the swarm
278                     member_votes[id] = i;
279                 }
280             }
281         }
282     }
283 }
284
285
286 </declaration>

```



```

287 <location id="id0" x="-136" y="195">
288   <name x="-127" y="170">Idle</name>
289 </location>
290 <location id="id1" x="272" y="195">
291   <name x="289" y="204">InSwarm</name>
292 </location>
293 <location id="id2" x="272" y="42">
294   <name x="221" y="8">Leader</name>
295 </location>
296 <location id="id3" x="654" y="-34">
297   <name x="612" y="-68">WaitingSwarm</name>
298 </location>
299 <location id="id4" x="629" y="195">
300   <name x="595" y="161">UpdatingLocation</name>
301 </location>
302 <init ref="id0"/>
303 <transition>
304   <source ref="id1"/>
305   <target ref="id1"/>
306   <label kind="guard" x="102" y="246">in_swarm(id)</label>
307   <label kind="synchronisation" x="85" y="263">member_election?</label>
308   <label kind="assignment" x="51" y="280">vote_member(id), vote_counter
      ++</label>
309   <nail x="178" y="255"/>
310   <nail x="229" y="280"/>
311 </transition>

```

```

312 <transition>
313   <source ref="id2"/>
314   <target ref="id1"/>
315   <label kind="guard" x="314" y="76">is_leader(id) == false</label>
316   <nail x="331" y="127"/>
317 </transition>
318 <transition>
319   <source ref="id1"/>
320   <target ref="id4"/>
321   <label kind="guard" x="323" y="153">reached_goal(id) == false & amp;
      amp; in_swarm(id)</label>
322   <label kind="synchronisation" x="366" y="170">update_location?</label>
323   <label kind="assignment" x="383" y="195">move(id)</label>
324   <nail x="341" y="195"/>
325 </transition>
326 <transition>
327   <source ref="id1"/>
328   <target ref="id1"/>
329   <label kind="guard" x="102" y="144">in_swarm(id)</label>
330   <label kind="synchronisation" x="178" y="119">election?</label>
331   <label kind="assignment" x="195" y="136">vote(id)</label>
332   <nail x="161" y="136"/>
333   <nail x="255" y="136"/>
334 </transition>
335 <transition>
336   <source ref="id1"/>

```

```

337     <target ref="id2"/>
338     <label kind="guard" x="25" y="68">is_leader(id)&amp;&amp;vote_counter
        ==Needed
339 &amp;&amp;updating_mission==true</label>
340     <label kind="assignment" x="93" y="51">update_leader_position(id)</label
        >
341     <nail x="272" y="153"/>
342     <nail x="272" y="127"/>
343 </transition>
344 <transition>
345     <source ref="id2"/>
346     <target ref="id0"/>
347     <label kind="synchronisation" x="0" y="17">mission_end?</label>
348     <nail x="0" y="42"/>
349     <nail x="-136" y="42"/>
350     <nail x="-136" y="161"/>
351 </transition>
352 <transition>
353     <source ref="id1"/>
354     <target ref="id0"/>
355     <label kind="synchronisation" x="34" y="204">mission_end?</label>
356     <nail x="161" y="229"/>
357     <nail x="161" y="229"/>
358     <nail x="-17" y="229"/>
359 </transition>
360 <transition>

```

```

361     <source ref="id3"/>
362     <target ref="id1"/>
363     <label kind="synchronisation" x="663" y="93">location.updated!</label>
364     <label kind="assignment" x="663" y="76">vote_counter = 0</label>
365     <nail x="654" y="340"/>
366     <nail x="314" y="340"/>
367 </transition>
368 <transition>
369     <source ref="id1"/>
370     <target ref="id0"/>
371     <label kind="guard" x="8" y="331">in_swarm(id) == false</label>
372     <label kind="synchronisation" x="25" y="306">update_status?</label>
373     <nail x="272" y="331"/>
374     <nail x="-136" y="331"/>
375 </transition>
376 <transition>
377     <source ref="id4"/>
378     <target ref="id1"/>
379     <label kind="synchronisation" x="399" y="246">location.updated?</label>
380     <nail x="629" y="272"/>
381     <nail x="322" y="272"/>
382 </transition>
383 <transition>
384     <source ref="id2"/>
385     <target ref="id3"/>

```

```

386      <label kind="guard" x="306" y="-93">swarm_reached_goal() == false &
          amp;& is_leader(id)
387 & amp;& updating_mission == true</label>
388      <label kind="synchronisation" x="382" y="-59">update_location!</label>
389      <label kind="assignment" x="416" y="-34">move(id)</label>
390      <nail x="271" y="-34"/>
391      <nail x="517" y="-34"/>
392  </transition>
393  <transition>
394      <source ref="id0"/>
395      <target ref="id1"/>
396      <label kind="guard" x="-42" y="153">in_swarm(id)</label>
397      <label kind="synchronisation" x="-51" y="170">update_status?</label>
398  </transition>
399  </template>
400  <template>
401      <name>MissionControl</name>
402      <declaration>
403  // function to reset array
404  void reset_arrays(){
405
406      for(i:int[0,N-1])
407      {
408          drone_candidates[i] = 0;
409          drone_time[i] = 0;
410      }

```

```

411 }
412
413
414 // check if leader position is the same as fire/target position
415     bool check_leader_pos(){
416         if(leader_position[0] == goal[0] && leader_position[1] == goal[1])
417             return true;
418         else
419             return false;
420     }
421
422 // Check if drone is in swarm
423     bool in_swarm(int id){
424         return drone_status[id] != 0;
425     }
426
427 // Find the current drone in the swarm with capability return its id
428     int find_drone_in_swarm(int capability){
429         for (id : int[0,N-1])
430             {
431                 if(drone_status[id] != 0 && drone_capability[id] == capability)
432                     {
433                         return id;
434                     }
435             }
436         return -1;

```

```

436     }
437
438 // Leader only stops issuing move commands if everybody reached location.
439 // Mission is only complete when swarm reached goal
440 bool swarm_reached_goal(){
441     bool reached = true;
442     for (id : int[0,N-1])
443     {
444         if(in_swarm(id))
445         {
446             reached = reached && drone_location_x[id] ==
447                     goal_location[0] && drone_location_y[id] ==
448                     goal_location[1];
449         }
450     }
451     return reached;
452 }
453
454 // linear_distance() function returns the calculated linear distance i.e the linear
455 // distance of the drone from the requesting node
456
457 int linear_distance(int goal[2], int a, int b) {
458     int x_coord = goal[0];
459     int y_coord = goal[1];
460
461     int xsquare = (x_coord-a)*(x_coord-a);

```

```

459     int ysquare = (y_coord-b)*(y_coord-b);
460     int sum1 = xsquare + ysquare;
461     sum1 = fint(sqrt(sum1));
462     return sum1; // fint fuction converts boolean or floating point value to
        integer
463 }
464
465
466
467
468 // This assumes leader drone is located at the GS when mission first starts
469 bool within_reach(int x, int y){
470     if( linear_distance(leader_position, x, y) &lt;= comm_reach)
471     {
472         return true;
473     }
474     else
475     {
476         return false;
477     }
478 }
479
480
481
482 // function that determines if the capability if needed for the mission
483 bool is_capabilities_needed(int capability)

```



```

484 {
485     for (i : int[0,2])
486     {
487         if(mission_capabilities[i] == capability) return true;
488     }
489     return false;
490 }
491
492
493 // maybe add a counter here that goes through capabilities needed in this mission
         and find possible members on each iteration of sim
494
495 int possible_member(){
496
497     for (i : int[0,N-1])
498     {
499         if(drone_status[i] == 0 && drone_battery[i] > 49 &&
                within_reach(drone_location_x[i], drone_location_y[i]) &&
                is_capabilities_needed(drone_capability[i]))
500         {
501             drone_candidates[i] = 1;
502             candidate_counter++;
503         }
504     }
505     return candidate_counter;
506 }

```

```

507
508 //checks the results of the votes for leader to see if consensus was achieved
509 bool voting_results(int check_votes[Needed]){
510
511     for (i : int[0,Needed-2])
512     {
513         if (check_votes[i] != check_votes[i + 1])
514             return true;
515     }
516     return false;
517 }
518
519
520 // based on the result of the election, this function assigns a leader drone
521 void elect_leader(){
522
523     for (i : int[0,Needed-1])
524     {
525         int cap = mission_capabilities[i];
526         id_voter = find_drone_in_swarm(cap);
527         check_votes[i]=votes[id_voter];
528
529     }
530
531     if(voting_results(check_votes)==false)
532     {

```

```

533     drone_status[votes[id_voter]] = 1;
534 }
535
536 voting_in_prog = false;
537 updating_mission = true;
538
539 }
540
541 // based on the result of the election, this function assigns a member drone
542 void elect_members(){
543
544     if(vote_counter == Needed)
545     {
546         for (i : int[0,Needed-1])
547         {
548             int cap = mission_capabilities[i];
549             id_voter = find_drone_in_swarm(cap);
550             check_member_votes[i] = member_votes[id_voter];
551         }
552
553         if(voting_results(check_member_votes)==false)
554         {
555             //get id of drone already in swarm with same capability as candidate to
556             join (id)
557             int curr_id = find_drone_in_swarm(drone_capability[member_votes[
558                 id_voter]]);

```

```

557         drone_status[curr_id] = 0;
558         drone_status[member_votes[id_voter]] = -1;
559
560     }
561
562     voting_in_prog = false;
563 }
564
565     updating_mission = true;
566     candidate_counter = 0;
567     vote_counter = 0;
568     reset_arrays();
569
570 }
571
572
573
574 </declaration>
575     <location id="id5" x="-272" y="195">
576         <name x="-289" y="161">Start</name>
577     </location>
578     <location id="id6" x="136" y="195">
579         <name x="93" y="204">ElectMembers</name>
580     </location>
581     <location id="id7" x="578" y="195">
582         <name x="552" y="161">MissionStarted</name>

```

```

583     </location>
584     <location id="id8" x="-93" y="195">
585         <name x="-144" y="161">CheckMembers</name>
586     </location>
587     <location id="id9" x="756" y="195">
588         <name x="688" y="161">MissionAccomplished</name>
589     </location>
590     <location id="id10" x="365" y="195">
591         <name x="323" y="221">LeaderElection</name>
592     </location>
593     <location id="id11" x="221" y="195">
594         <name x="170" y="221">UpdateMembers</name>
595     </location>
596     <init ref="id5"/>
597     <transition>
598         <source ref="id11"/>
599         <target ref="id10"/>
600         <label kind="synchronisation" x="246" y="178">update_status!</label>
601         <label kind="assignment" x="229" y="204">updating_mission = true</
            label>
602     </transition>
603     <transition>
604         <source ref="id6"/>
605         <target ref="id6"/>
606         <label kind="guard" x="25" y="85">updating_mission==true</label>
607         <label kind="synchronisation" x="43" y="68">member_election!</label>

```

```

608     <label kind="assignment" x="26" y="51">updating_mission=false</label>
609     <nail x="94" y="153"/>
610     <nail x="51" y="110"/>
611     <nail x="102" y="110"/>
612     <nail x="161" y="110"/>
613 </transition>
614 <transition>
615     <source ref="id10"/>
616     <target ref="id7"/>
617     <label kind="guard" x="391" y="170">vote_counter == Needed</label>
618     <label kind="synchronisation" x="433" y="212">update_status!</label>
619     <label kind="assignment" x="416" y="195">elect_leader()</label>
620 </transition>
621 <transition>
622     <source ref="id10"/>
623     <target ref="id10"/>
624     <label kind="guard" x="246" y="51">vote_counter == 0 & amp; & amp;
        updating_mission == true</label>
625     <label kind="synchronisation" x="348" y="68">election!</label>
626     <label kind="assignment" x="297" y="93">updating_mission = false</label>
        >
627     <nail x="281" y="93"/>
628     <nail x="408" y="93"/>
629     <nail x="467" y="93"/>
630 </transition>
631 <transition>

```

```

632     <source ref="id7"/>
633     <target ref="id9"/>
634     <label kind="guard" x="594" y="195">swarm_reached_goal()</label>
635     <label kind="synchronisation" x="636" y="212">mission_end!</label>
636     <label kind="assignment" x="603" y="229">updating_mission = 0</label>
637 </transition>
638 <transition>
639     <source ref="id8"/>
640     <target ref="id6"/>
641     <label kind="guard" x="-59" y="178">updating_mission == true</label>
642     <label kind="synchronisation" x="-25" y="161">update_status!</label>
643     <label kind="assignment" x="-34" y="195">possible_member()</label>
644 </transition>
645 <transition>
646     <source ref="id5"/>
647     <target ref="id8"/>
648     <label kind="assignment" x="-263" y="212">goal_location[0] = goal[0],
649 goal_location[1] = goal[1]</label>
650 </transition>
651 <transition>
652     <source ref="id7"/>
653     <target ref="id8"/>
654     <label kind="synchronisation" x="178" y="271">location_updated?</label>
655     <label kind="assignment" x="68" y="289">updating_mission = true,
        vote_counter=0, reset_arrays()</label>
656 <nail x="578" y="263"/>

```

```

657      <nail x="-93" y="263"/>
658  </transition>
659  <transition>
660      <source ref="id6"/>
661      <target ref="id11"/>
662      <label kind="guard" x="161" y="144">updating_mission == false</label>
663      <label kind="assignment" x="153" y="161">elect_members()</label>
664  </transition>
665  </template>
666  <system>// System declarations
667
668  //drone instances
669  Drone0 = Drone(0, drone_capability[0], drone_battery[0]);
670  Drone1 = Drone(1, drone_capability[1], drone_battery[1]);
671  Drone2 = Drone(2, drone_capability[2], drone_battery[2]);
672  Drone3 = Drone(3, drone_capability[3], drone_battery[3]);
673  Drone4 = Drone(4, drone_capability[4], drone_battery[4]);
674  Drone5 = Drone(5, drone_capability[5], drone_battery[5]);
675  Drone6 = Drone(6, drone_capability[6], drone_battery[6]);
676  Drone7 = Drone(7, drone_capability[7], drone_battery[7]);
677
678  //mission instance
679  MissionC = MissionControl();
680
681  //system definition

```



```

682 system MissionC, Drone0, Drone1, Drone2, Drone3, Drone4, Drone5, Drone6, Drone7
        ;
683
684
685
686
687 </system>
688 <queries>
689 <query>
690 <formula>A[] (vote_counter == Needed || vote_counter == 0)</formula>
691 <comment>The number of votes is always equal to 0 (if no votation occurs)
        or 3 which is the needed votes (there are always 3 drones in the swarm)</
        comment>
692 </query>
693 <query>
694 <formula>A[] deadlock imply (MissionC.MissionAccomplished)</formula>
695 <comment>if a deadlock exists anywhere it means that mission was
        accomplished</comment>
696 </query>
697 <query>
698 <formula>A[] ((drone_status[0] + drone_status[1] + drone_status[2] +
        drone_status[3] + drone_status[4] + drone_status[5] + drone_status[6] +
        drone_status[7]) <= 0)</formula>
699 <comment>This property checks that there is never more than one leader in
        the swarm. Drone status is represented as -1 for in swarm, 0 for idle, and
        1 for leader. Therefore, if the total

```

700 number of drones in the swarm at any point is 3, the sum of their status values must
be negative to respect the one leader policy.

701 </comment>

702 </query>

703 <query>

704 <formula> $A[]((\text{abs}(\text{drone_status}[0]) + \text{abs}(\text{drone_status}[1]) + \text{abs}(\text{drone_status}[2]) + \text{abs}(\text{drone_status}[3]) + \text{abs}(\text{drone_status}[4]) + \text{abs}(\text{drone_status}[5]) + \text{abs}(\text{drone_status}[6]) + \text{abs}(\text{drone_status}[7])) == \text{Needed})$ </formula>

705 <comment>This property complements the property verified above.

Considering again that drone status -1 means in swarm, 0 means idle, and
 1 means leader: the number of drones in the swarm

706 at any point of the missions is equal to the absolute value of the sum of their status
values. This property verifies that always globally there are 3 drones in the
swarm. </comment>

707 </query>

708 <query>

709 <formula> $A[] \text{ forall}(i:\text{int}[0,7])(\text{drone_status}[i] == 1 \text{ imply drone_capability}[i] == 1)$ </formula>

710 <comment>property that verifies if any drone is a leader then its capability is
communicator </comment>

711 </query>

712 <query>

713 <formula> $A\<\>\text{ drone_status}[0] == 0 \text{ imply Drone0.Idle}$ </formula>

714 <comment>This property verifies that if a drone is not part of the swarm
anymore then it always eventually goes to state Idle.</comment>

715 </query>

```

716    <query>
717        <formula>A[] forall(i:int[0,7])(drone_status[i] == -1 || drone_status[i] == 1)
            imply drone_battery[i] >= 49</formula>
718        <comment>This property verifies that drones in the swarm must have at least
            49 battery to be part of the swarm. </comment>
719    </query>
720 </queries>
721 </nta>

```

CodeFiles/Swarm_consensus_final.xml

Appendix C

ROS Base Classes

```
1 #include <bits/stdc++.h>
2 #include <gnc_functions.hpp>
3 using namespace std;
4
5 class Drone
6 {
7 private:
8     int x;
9     int y;
10    int i;
11    ros::NodeHandle nh;
12
13 public:
14
15    Drone(int i)
```

```

16  {
17      // added
18      this->i = i;
19  }
20  bool is_leader(int id)
21  {
22      vector<int> drone_status;
23      nh.getParam("/drone_status", drone_status);
24      return (drone_status[id] == 1);
25  }
26  bool in_swarm(int id)
27  {
28      vector<int> drone_status;
29      nh.getParam("/drone_status", drone_status);
30      return drone_status[id] != 0;
31  }
32  int find_drone_in_swarm(int capability)
33  {
34      vector<int> drone_capability;
35      nh.getParam("/drone_capability", drone_capability);
36      int N;
37      nh.getParam("/N", N);
38      vector<int> drone_status;
39      nh.getParam("/drone_status", drone_status);
40      for (int id = 0; id <= (N - 1); id++)
41      {

```

```

42     if ((drone_status[id] != 0) && (drone_capability[id] == capability))
43     {
44         return id;
45     }
46 }
47
48     return -1;
49 }
50 bool swarm_reached_goal()
51 {
52     vector<int> goal_location;
53     nh.getParam("/goal_location", goal_location);
54     int N;
55     nh.getParam("/N", N);
56     vector<int> drone_location_y;
57     nh.getParam("/drone_location_y", drone_location_y);
58     vector<int> drone_location_x;
59     nh.getParam("/drone_location_x", drone_location_x);
60     bool reached = true;
61     for (int id = 0; id <= (N - 1); id++)
62     {
63         if (in_swarm(id))
64         {
65             reached = (reached && (drone_location_x[id] == goal_location[0])) && (
66                 drone_location_y[id] == goal_location[1]);

```

```

67     }
68
69     return reached;
70 }
71 void update_leader_position(int id)
72 {
73     vector<int> leader_position;
74     nh.getParam("/leader_position", leader_position);
75     vector<int> drone_location_x;
76     nh.getParam("/drone_location_x", drone_location_x);
77     vector<int> drone_location_y;
78     nh.getParam("/drone_location_y", drone_location_y);
79     leader_position[0] = drone_location_x[id];
80     leader_position[1] = drone_location_y[id];
81     nh.setParam("/leader_position", leader_position);
82 }
83 bool reached_goal(int id)
84 {
85     vector<int> goal_location;
86     nh.getParam("/goal_location", goal_location);
87     vector<int> drone_location_x;
88     nh.getParam("/drone_location_x", drone_location_x);
89     vector<int> drone_location_y;
90     nh.getParam("/drone_location_y", drone_location_y);
91     return (drone_location_x[id] == goal_location[0]) && (drone_location_y[id] ==
        goal_location[1]);

```

```

92  }
93  int linear_distance(vector<int> goal, int a, int b)
94  {
95      //vector<int> goal;
96      //nh.getParam("/goal", goal);
97
98      int x_coord = goal[0];
99      int y_coord = goal[1];
100     int xsquare = (x_coord - a) * (x_coord - a);
101     int ysquare = (y_coord - b) * (y_coord - b);
102     int sum1 = xsquare + ysquare;
103     sum1 = int(sqrt(sum1));
104     return sum1;
105 }
106 void calculate_drone_times(int cap)
107 {
108     vector<int> drone_capability;
109     nh.getParam("/drone_capability", drone_capability);
110     vector<int> drone_time;
111     nh.getParam("/drone_time", drone_time);
112     int N;
113     nh.getParam("/N", N);
114     int dist;
115     nh.getParam("/dist", dist);
116     vector<int> leader_position;
117     nh.getParam("/leader_position", leader_position);

```



```

118     vector<int> drone_location_y;
119     nh.getParam("/drone_location_y", drone_location_y);
120     vector<int> drone_location_x;
121     nh.getParam("/drone_location_x", drone_location_x);
122     vector<int> time_to_swarm;
123     nh.getParam("/time_to_swarm", time_to_swarm);
124     vector<int> drone_candidates;
125     nh.getParam("/drone_candidates", drone_candidates);
126     for (int i = 0; i <= (N - 1); i++)
127     {
128         if ((drone_candidates[i] == 1) && (drone_capability[i] == cap))
129         {
130             dist = linear_distance(leader_position, drone_location_x[i], drone_location_y[
131                                     i]);
132             if ((dist > 0) && (dist < 9))
133             {
134                 drone_time[i] = time_to_swarm[dist - 1];
135                 nh.setParam("/drone_time", drone_time);
136             }
137         }
138     }
139 }
140 void move(int id)
141 {
142     vector<int> goal_location;

```

```

143     nh.getParam("/goal_location", goal_location);
144     vector<int> drone_location_y;
145     nh.getParam("/drone_location_y", drone_location_y);
146     vector<int> drone_battery;
147     nh.getParam("/drone_battery", drone_battery);
148     vector<int> drone_location_x;
149     nh.getParam("/drone_location_x", drone_location_x);
150     if (drone_location_x[id] != goal_location[0])
151     {
152         drone_location_x[id] = drone_location_x[id] + 1;
153         drone_battery[id] -= 1;
154         nh.setParam("/drone_location_x", drone_location_x);
155         nh.setParam("/drone_battery", drone_battery);
156     }
157     else
158     {
159         if (drone_location_y[id] != goal_location[1]){
160             drone_location_y[id] = drone_location_y[id] + 1;
161             drone_battery[id] -= 1;
162             nh.setParam("/drone_location_y", drone_location_y);
163             nh.setParam("/drone_battery", drone_battery);
164         }
165     }
166     if (is_leader(id))
167         update_leader_position(id);
168

```

```

169     x = drone_location.x[id];
170     y = drone_location.y[id];
171
172 }
173 void vote(int id)
174 {
175     vector<int> drone_capability;
176     nh.getParam("/drone_capability", drone_capability);
177     int vote_counter;
178     nh.getParam("/vote_counter", vote_counter);
179     int N;
180     nh.getParam("/N", N);
181     vector<int> votes;
182     nh.getParam("/votes", votes);
183     vector<int> drone_status;
184     nh.getParam("/drone_status", drone_status);
185     for (int i = 0; i <= (N - 1); i++)
186     {
187         if ((drone_status[i] == (-1)) && (drone_capability[i] == 1))
188         {
189             votes[id] = i;
190             nh.setParam("/votes", votes);
191         }
192     }
193     vote_counter++;
194     nh.setParam("/vote_counter", vote_counter);

```

```

195     }
196
197
198     int find_min(vector<int> arr, int capability)
199     {
200         vector<int> drone_capability;
201         nh.getParam("/drone_capability", drone_capability);
202         int N;
203         nh.getParam("/N", N);
204         vector<int> drone_candidates;
205         nh.getParam("/drone_candidates", drone_candidates);
206         int min = 100;
207         for (int i = 0; i <= (N - 1); i++)
208         {
209             if ((drone_capability[i] == capability) && (drone_candidates[i] == 1))
210             {
211                 if ((arr[i] != 0) && (arr[i] < min))
212                 {
213                     min = arr[i];
214                 }
215             }
216         }
217
218         return min;
219     }
220     void vote_member(int id)

```

```

221  {
222      vector<int> drone_capability;
223      nh.getParam("/drone_capability", drone_capability);
224      int min_time;
225      nh.getParam("/min_time", min_time);
226      int N;
227      nh.getParam("/N", N);
228      vector<int> drone_time;
229      nh.getParam("/drone_time", drone_time);
230      vector<int> member_votes;
231      nh.getParam("/member_votes", member_votes);
232      vector<int> drone_battery;
233      nh.getParam("/drone_battery", drone_battery);
234      vector<int> drone_candidates;
235      nh.getParam("/drone_candidates", drone_candidates);
236      for (int i = 0; i <= (N - 1); i++)
237      {
238          if (drone_candidates[i] == 1)
239          {
240              int current_drone = find_drone_in_swarm(drone_capability[i]);
241              calculate_drone_times(drone_capability[i]);
242              min_time = find_min(drone_time, drone_capability[current_drone]);
243              if (drone_time[i] == min_time)
244              {
245                  if (((drone_battery[current_drone] < 50) && (drone_battery[i] >
                      drone_battery[current_drone])))

```

```

246         {
247             member_votes[id] = i;
248             nh.setParam("/member_votes", member_votes);
249         }
250     }
251 }
252 }
253
254 }
255 };

```

CodeFiles/Drone_base_class.cpp

```

1 #include <bits/stdc++.h>
2 #include <gnc_functions.hpp>
3 using namespace std;
4
5 class MissionControl
6 {
7 private:
8
9     ros::NodeHandle nh;
10
11 public:
12     MissionControl()
13     {
14

```

```

15  }
16  void reset_arrays()
17  {
18      vector<int> drone_time;
19      nh.getParam("/drone_time", drone_time);
20      int N;
21      nh.getParam("/N", N);
22      vector<int> drone_candidates;
23      nh.getParam("/drone_candidates", drone_candidates);
24      for (int i = 0; i <= (N - 1); i++)
25      {
26          drone_candidates[i] = 0;
27          nh.setParam("/drone_candidates", drone_candidates);
28          drone_time[i] = 0;
29          nh.setParam("/drone_time", drone_time);
30      }
31
32  }
33  bool check_leader_pos()
34  {
35      vector<int> leader_position;
36      nh.getParam("/leader_position", leader_position);
37      vector<int> goal;
38      nh.getParam("/goal", goal);
39      if ((leader_position[0] == goal[0]) && (leader_position[1] == goal[1]))
40          return true;

```

```

41     else
42         return false;
43 }
44 bool in_swarm(int id)
45 {
46     vector<int> drone_status;
47     nh.getParam("/drone_status", drone_status);
48     return drone_status[id] != 0;
49
50 }
51 int find_drone_in_swarm(int capability)
52 {
53     vector<int> drone_capability;
54     nh.getParam("/drone_capability", drone_capability);
55     int N;
56     nh.getParam("/N", N);
57     vector<int> drone_status;
58     nh.getParam("/drone_status", drone_status);
59     for (int id = 0; id <= (N - 1); id++)
60     {
61         if ((drone_status[id] != 0) && (drone_capability[id] == capability))
62         {
63             return id;
64         }
65     }
66

```



```

67     return -1;
68 }
69 bool swarm_reached_goal()
70 {
71     vector<int> goal_location;
72     nh.getParam("/goal_location", goal_location);
73     int N;
74     nh.getParam("/N", N);
75     vector<int> drone_location_y;
76     nh.getParam("/drone_location_y", drone_location_y);
77     vector<int> drone_location_x;
78     nh.getParam("/drone_location_x", drone_location_x);
79     bool reached = true;
80     for (int id = 0; id <= (N - 1); id++)
81     {
82         if (in_swarm(id))
83         {
84             reached = (reached && (drone_location_x[id] == goal_location[0])) && (
                        drone_location_y[id] == goal_location[1]);
85         }
86     }
87
88     return reached;
89 }
90 int linear_distance(vector<int> goal, int a, int b)
91 {

```

```

92     //vector<int> goal;
93     //nh.getParam("/goal", goal); does not like same name for argument and
        variable
94     int x_coord = goal[0];
95     int y_coord = goal[1];
96     int xsquare = (x_coord - a) * (x_coord - a);
97     int ysquare = (y_coord - b) * (y_coord - b);
98     int sum1 = xsquare + ysquare;
99     sum1 = int(sqrt(sum1)); // casting to integer
100     return sum1;
101 }
102 bool within_reach(int x,int y)
103 {
104     vector<int> leader_position;
105     nh.getParam("/leader_position", leader_position);
106     int comm_reach;
107     nh.getParam("/comm_reach", comm_reach);
108     if (linear_distance(leader_position, x, y) <= comm_reach)
109     {
110         return true;
111     }
112     else
113     {
114         return false;
115     }
116 }

```

```

117  bool is_capabilities_needed(int capability)
118  {
119      vector<int> mission_capabilities;
120      nh.getParam("/mission_capabilities", mission_capabilities);
121      for (int i = 0; i <= 2; i++)
122      {
123          if (mission_capabilities[i] == capability)
124              return true;
125      }
126
127      return false;
128  }
129  int possible_member()
130  {
131      vector<int> drone_capability;
132      nh.getParam("/drone_capability", drone_capability);
133      int N;
134      nh.getParam("/N", N);
135      vector<int> drone_status;
136      nh.getParam("/drone_status", drone_status);
137      vector<int> drone_location_y;
138      nh.getParam("/drone_location_y", drone_location_y);
139      vector<int> drone_battery;
140      nh.getParam("/drone_battery", drone_battery);
141      vector<int> drone_location_x;
142      nh.getParam("/drone_location_x", drone_location_x);

```

```

143  int candidate_counter;
144  nh.getParam("/candidate_counter", candidate_counter);
145  vector<int> drone_candidates;
146  nh.getParam("/drone_candidates", drone_candidates);
147  for (int i = 0; i <= (N - 1); i++)
148  {
149      if (((drone_status[i] == 0) && (drone_battery[i] > 49)) && within_reach(
150          drone_location_x[i], drone_location_y[i])) && is_capabilities_needed(
151              drone_capability[i]))
152      {
153          drone_candidates[i] = 1; // these need to get set on the parameter server
154          candidate_counter++;
155          nh.setParam("/drone_candidates", drone_candidates);
156          nh.setParam("/candidate_counter", candidate_counter);
157      }
158  }
159  return candidate_counter;
160 }
161 bool voting_results(vector<int> check_votes) // modified argument of function
162 {
163     int Needed;
164     nh.getParam("/Needed", Needed);
165     //vector<int> check_votes;
166     //nh.getParam("/check_votes", check_votes);
167     for (int i = 0; i <= (Needed - 2); i++)

```

```

167     {
168         if (check_votes[i] != check_votes[i + 1])
169             return true;
170     }
171
172     return false;
173 }
174 void elect_leader()
175 {
176     vector<int> votes;
177     nh.getParam("/votes", votes);
178     vector<int> drone_status;
179     nh.getParam("/drone_status", drone_status);
180     int voting_in_prog;
181     nh.getParam("/voting_in_prog", voting_in_prog);
182     vector<int> mission_capabilities;
183     nh.getParam("/mission_capabilities", mission_capabilities);
184     int Needed;
185     nh.getParam("/Needed", Needed);
186     int updating_mission;
187     nh.getParam("/updating_mission", updating_mission);
188     vector<int> check_votes;
189     nh.getParam("/check_votes", check_votes);
190     int id_voter;
191     nh.getParam("/id_voter", id_voter);
192     for (int i = 0; i <= (Needed - 1); i++)

```

```

193     {
194         int cap = mission_capabilities[i];
195         id_voter = find_drone_in_swarm(cap);
196         check_votes[i] = votes[id_voter];
197         nh.setParam("/check_votes", check_votes);
198     }
199
200     if (voting_results(check_votes) == false)
201     {
202         drone_status[votes[id_voter]] = 1;
203         nh.setParam("/drone_status", drone_status);
204     }
205     voting_in_prog = false;
206     nh.setParam("/voting_in_prog", voting_in_prog);
207     updating_mission = true;
208     nh.setParam("/updating_mission", updating_mission);
209 }
210 void elect_members()
211 {
212     int voting_in_prog;
213     nh.getParam("/voting_in_prog", voting_in_prog);
214     int Needed;
215     nh.getParam("/Needed", Needed);
216     int candidate_counter;
217     nh.getParam("/candidate_counter", candidate_counter);
218     int id_voter;

```

```

219     nh.getParam("/id_voter", id_voter);
220     vector<int> member_votes;
221     nh.getParam("/member_votes", member_votes);
222     vector<int> drone_status;
223     nh.getParam("/drone_status", drone_status);
224     int vote_counter;
225     nh.getParam("/vote_counter", vote_counter);
226     vector<int> check_member_votes;
227     nh.getParam("/check_member_votes", check_member_votes);
228     vector<int> drone_capability;
229     nh.getParam("/drone_capability", drone_capability);
230     vector<int> mission_capabilities;
231     nh.getParam("/mission_capabilities", mission_capabilities);
232     int updating_mission;
233     nh.getParam("/updating_mission", updating_mission);
234     if (vote_counter == Needed)
235     {
236         for (int i = 0; i <= (Needed - 1); i++)
237         {
238             int cap = mission_capabilities[i];
239             id_voter = find_drone_in_swarm(cap);
240             check_member_votes[i] = member_votes[id_voter];
241             nh.setParam("/check_member_votes", check_member_votes);
242         }
243
244         if (voting_results(check_member_votes) == false)

```

```

245     {
246         int curr_id = find_drone_in_swarm(drone_capability[member_votes[id_voter
                ]]);
247         drone_status[curr_id] = 0;
248         drone_status[member_votes[id_voter]] = -1;
249         nh.setParam("/drone_status", drone_status);
250     }
251     voting_in_prog = false;
252     nh.setParam("/voting_in_prog", voting_in_prog);
253 }
254 updating_mission = true;
255 nh.setParam("/updating_mission", updating_mission);
256 candidate_counter = 0;
257 nh.setParam("/candidate_counter", candidate_counter);
258 vote_counter = 0;
259 nh.setParam("/vote_counter", vote_counter);
260 reset_arrays();
261 }
262 };

```

CodeFiles/MissionControl_base_class.cpp

Appendix D

ROS Node Control Files

```
1 #include "base_classes/Drone_base_class.cpp"
2 #include "ros/ros.h"
3 #include <thread>
4 #include <chrono>
5 #include <std_msgs/Int8.h>
6
7 using namespace std;
8
9 // locally global variables to use in callback functions
10 int update_status_var;
11 int member_election_var;
12 int leader_election_var;
13 int update_location_var;
14 int mission_end_var;
15
```

```

16 // same as mission control template
17 int location_updated_var;
18
19 void locationUpdatedCallback(std_msgs::Int8 location_updated){
20     location_updated_var = location_updated.data;
21 }
22
23
24 void statusCallback(std_msgs::Int8 update_status){
25     update_status_var = update_status.data;
26 }
27
28 void memberElectionCallback(std_msgs::Int8 member_election){
29     member_election_var = member_election.data;
30 }
31
32 void leaderElectionCallback(std_msgs::Int8 leader_election){
33     leader_election_var = leader_election.data;
34 }
35
36 void updateLocationCallback(std_msgs::Int8 update_location){
37     update_location_var = update_location.data;
38 }
39
40 void missionEndCallback(std_msgs::Int8 mission_end){
41     mission_end_var = mission_end.data;

```

```

42 }
43
44
45
46 int main(int argc, char** argv)
47 {
48
49
50     // parse the argument passed in launch file to represent current drone id
51     int id = atoi(argv[1]);
52
53     // private node handle
54     // initialize ROS
55     ros::init(argc, argv, "drone_node");
56     ros::NodeHandle nh("~");
57
58     // instantiate Drone class
59     Drone* ThisDrone = new Drone(id);
60
61
62
63     // This function takes our ros node handle as an input and initializes subscribers
64     // that will collect the necessary information from our autopilot.
65     // @returns n/a
66     init_publisher_subscriber(nh);
67

```

```

68  // wait for FCU connection
69  wait4connect();
70
71  // changing mode to GUIDED
72  set_mode("GUIDED");
73
74  //create local reference frame
75  initialize_local_frame();
76
77  std::string ThisNamespace;
78  nh.getParam("namespace", ThisNamespace);
79  ROS_INFO("THIS NAMESPACE IS: %s", ThisNamespace.c_str());
80
81  // define subscribers
82  ros::Subscriber update_status_sub;
83  ros::Subscriber member_election_sub;
84  ros::Subscriber leader_election_sub;
85  ros::Subscriber update_location_sub;
86  ros::Subscriber location_updated_sub;
87  ros::Subscriber mission_end_sub;
88
89  //Publishers
90  ros::Publisher location_updated_pub = nh.advertise<std_msgs::Int8>(" /
    location_updated", 1, true); // checking without namespace and adding latch
91  ros::Publisher update_location_pub = nh.advertise<std_msgs::Int8>(" /
    update_location", 1, true);

```

```

92
93 // local global variable to get and set vote counter variable
94 int vote_counter;
95
96 int Needed;
97
98 int updating_mission;
99
100 //define standard sybc msg
101 std_msgs::Int8 sync;
102 sync.data = 1;
103
104 // define TA states as enum
105 enum STATES
106 {
107     Idle, InSwarm, Leader, UpdatingLocation, WaitingSwarm
108
109 } STATE;
110
111 STATE = Idle;
112
113 // create a flag to know when the drones voted when InSwarm state
114 int flag = 0;
115
116 ROS_INFO("Drones going into while loop");
117

```

```

118     ros::Rate rate(0.5);
119
120     while(ros::ok()){
121
122         ros::spinOnce();
123
124         switch(STATE){
125
126             case Idle:
127                 {
128
129                     // idle to in swarm
130                     if(ThisDrone->in_swarm(id)){
131
132                         update_status_sub = nh.subscribe("/update_status", 1,
133                             statusCallback);
134                         while(update_status_var != 1){
135                             ros::spinOnce();
136                         }
137
138                         if(update_status_var == 1){
139                             ROS_INFO("Inside Idle state, Drones should takeoff");
140                             takeoff(10);
141                             ROS_INFO("Waiting for drones to reach waypoint");
142                             STATE = InSwarm;
143                             update_status_sub.shutdown();

```

```

143         update_status_var = 0;
144     }
145
146 }
147
148     rate.sleep();
149     break;
150 }
151
152 case InSwarm:
153 {
154     // member election logic
155     ROS_INFO("Drones InSwarm case");
156     if(flag == 0 && ThisDrone->in_swarm(id)){
157
158         ROS_INFO("Drones in election loops");
159         member_election_sub = nh.subscribe("/member_election", 1,
160             memberElectionCallback);
161         while(member_election_var != 1){
162             break;
163         }
164         nh.getParam("/vote_counter", vote_counter);
165         if(member_election_var == 1 && vote_counter < 3){ // vote
166             counter restriction added here

```

```

167         nh.getParam("/vote_counter", vote_counter);
168         ROS_INFO("Drones voted for members: %d", vote_counter);
169         STATE = InSwarm;
170         member_election_sub.shutdown();
171         member_election_var = 0;
172         nh.getParam("/vote_counter", vote_counter);
173
174     }
175
176     while(vote_counter != 0){
177         // wait till member election sets votes back to 0
178         nh.getParam("/vote_counter", vote_counter);
179     }
180
181     ROS_INFO("drones out of member election, moving to leader
        election");
182     nh.getParam("/vote_counter", vote_counter);
183     leader_election_sub = nh.subscribe("/leader_election", 1,
        leaderElectionCallback);
184     while(leader_election_var != 1){
185         break;
186     }
187     if(leader_election_var == 1 && vote_counter < 3){
188         ThisDrone->vote(id);
189         ros::Duration(3).sleep();
190         nh.getParam("/vote_counter", vote_counter);

```



```

191         ROS_INFO("Drones voted for leader: %d", vote_counter);
192         STATE = InSwarm;
193         leader_election_sub.shutdown();
194         leader_election_var = 0;
195         flag = 1;
196         break; // bc leader drone at this point needs to go to leader
                state
197
198     }
199
200 }
201
202 // transition to leader
203 nh.getParam("/vote_counter", vote_counter);
204 nh.getParam("/updating_mission", updating_mission);
205 if(ThisDrone->is_leader(id) && vote_counter == Needed &&
        updating_mission == 1){ //inSwarm to leader transition
206     ThisDrone->update_leader_position(id);
207     STATE = Leader;
208     ROS_INFO("leader transitions");
209     rate.sleep();
210     break;
211 }
212
213 //transition to updating location
214 if(flag == 1 && ThisDrone->in_swarm(id) == true && ThisDrone

```

```

215         ->reached_goal(id) == false){
216
217         update_location_sub = nh.subscribe("/update_location", 1,
218             updateLocationCallback);
219         while(update_location_var != 1){
220             ros::spinOnce();
221         }
222         if(update_location_var == 1){
223             ROS_INFO("Drone moves 1 unit");
224             ThisDrone->move(id);
225             vector<int> drone_location_x;
226             vector<int> drone_location_y;
227             nh.getParam("/drone_location_x", drone_location_x);
228             nh.getParam("/drone_location_y", drone_location_y);
229             set_destination(drone_location_x[id], drone_location_y[id], 10,
230                 10);
231             ROS_INFO("Waiting for drones to reach waypoint");
232             STATE = UpdatingLocation;
233             update_location_sub.shutdown();
234             update_location_var = 0;
235             rate.sleep();
236             flag = 0;
237             break;
238         }
239     }
240 }
241

```

```

238         // InSwarm to Idle transition
239         if(ThisDrone->in_swarm(id) == false){
240             update_status_sub = nh.subscribe("/update_status", 1,
                statusCallback);
241             if(update_status_var == 1){
242                 land();
243                 STATE = Idle;
244                 update_status_sub.shutdown();
245                 update_status_var = 0;
246                 rate.sleep();
247                 break;
248             }
249
250         }
251
252         // Inswarm to idle transition
253         mission_end_sub = nh.subscribe("/mission_end", 1,
                missionEndCallback);
254         if(mission_end_var == 1){
255             land();
256             STATE = Idle;
257             mission_end_sub.shutdown();
258             mission_end_var = 0;
259             rate.sleep();
260             break;
261         }

```

```

262
263         ROS_INFO("breaking from inswarm case now");
264         rate.sleep();
265         break;
266     }
267
268     case Leader:
269     {
270         // leader to in swarm state transition
271         if(ThisDrone->is_leader(id) == false){
272             STATE = InSwarm;
273             rate.sleep();
274             break;
275         }
276
277         // leader to idle transition
278         mission_end_sub = nh.subscribe("/mission_end", 1,
                missionEndCallback);
279         if(mission_end_var == 1){
280             land();
281             STATE = Idle;
282             mission_end_sub.shutdown();
283             rate.sleep();
284             break;
285         }
286

```

```

287         // leader to waitingswarm transition
288         nh.getParam("/updating_mission", updating_mission);
289         if(ThisDrone->swarm_reached_goal() == false && ThisDrone->
            is_leader(id) && updating_mission == true){
290             while(update_location_pub.getNumSubscribers() < 2){
291                 ROS_INFO("waiting for mission control update location sub
                    ");
292             }
293             update_location_pub.publish(sync);
294             ROS_INFO("Leader Drone Moves 1 unit");
295             ThisDrone->move(id);
296             vector<int> drone_location_x;
297             vector<int> drone_location_y;
298             nh.getParam("/drone_location_x", drone_location_x);
299             nh.getParam("/drone_location_y", drone_location_y);
300             set_destination(drone_location_x[id], drone_location_y[id], 10, 10);
301             ROS_INFO("Waiting for drones to reach waypoint");
302             STATE = WaitingSwarm;
303             rate.sleep();
304             break;
305         }
306         rate.sleep();
307         break;
308     }
309
310     case UpdatingLocation:

```

```

311     {
312         //updating location to in swarm transition
313         ROS_INFO("Drone updating location");
314         location_updated_sub = nh.subscribe("/location_updated", 1,
            locationUpdatedCallback);
315         while(location_updated_var != 1){
316             ros::spinOnce();
317         }
318         if(location_updated_var == 1){
319             STATE = InSwarm;
320             location_updated_sub.shutdown();
321             rate.sleep();
322             break;
323         }
324         rate.sleep();
325         break;
326     }
327
328     case WaitingSwarm:
329     {
330
331         while(location_updated_pub.getNumSubscribers() < 2){
332             ROS_INFO("waiting for mission control location updated sub");
333         }
334
335         location_updated_pub.publish(sync);

```

```

336         vote_counter = 0;
337         nh.setParam("/vote_counter", vote_counter);
338         STATE = InSwarm;
339         // return flag to 0 for leader so it goes into election loop as well.
340         flag = 0;
341         rate.sleep();
342         break;
343     }
344 }
345
346     rate.sleep();
347 }
348     ROS_INFO("ROS IS NOT OKAY");
349 }

```

CodeFiles/Drone.cpp

```

1  #include "base_classes/MissionControl_base_class.cpp"
2  #include "ros/ros.h"
3  #include <thread>
4  #include <chrono>
5  #include <std_msgs/Int8.h>
6
7  using namespace std;
8
9  int location_updated_var;
10

```

```

11 void locationUpdatedCallback(std_msgs::Int8 location_updated){
12     location_updated_var = location_updated.data;
13 }
14
15 int main(int argc, char** argv)
16 {
17
18     // initialize ROS
19     ros::init(argc, argv, "MissionControl_node");
20     // private node handle
21     ros::NodeHandle nh("");
22
23     init_publisher_subscriber(nh);
24
25
26     MissionControl* MissionController = new MissionControl();
27
28     std::string ThisNamespace;
29     nh.getParam("namespace", ThisNamespace);
30
31     // all publishers — each represents a channel in uppaal
32     ros::Publisher update_status_pub = nh.advertise<std_msgs::Int8>((
        ThisNamespace+"/update_status").c_str(), 1, true);
33     ros::Publisher member_election_pub = nh.advertise<std_msgs::Int8>((
        ThisNamespace+"/member_election").c_str(), 1, true);
34     ros::Publisher election_pub = nh.advertise<std_msgs::Int8>((ThisNamespace+"/

```



```

        leader_election").c_str(), 1, true);
35  ros::Publisher mission_end_pub = nh.advertise<std_msgs::Int8>((ThisNamespace
        +"/mission_end").c_str(), 1, true);
36
37  //subscribers
38  ros::Subscriber location_updated_sub;
39
40  // define TA states as enum
41  enum STATES
42  {
43      Start, CheckMembers, ElectMembers, UpdateMembers, LeaderElection,
        MissionStarted, MissionAccomplished
44  } STATE;
45
46  STATE = Start;
47
48  int updating_mission;
49  int vote_counter;
50  int Needed;
51  vector<int> goal;
52
53  //define standard msg
54  std_msgs::Int8 sync;
55  sync.data = 1;
56
57

```

```

58     ros::Duration(10).sleep();
59
60     ROS_INFO("Mission Control going into while loop");
61
62     ros::Rate rate(0.5);
63
64     while(ros::ok()){
65
66         ros::spinOnce();
67
68         switch(STATE){
69
70             case Start:
71                 {
72                     // start to check members transition
73                     ROS_INFO("MissionControl in Start state");
74                     nh.getParam("/goal", goal);
75                     nh.setParam("/goal_location", goal);
76                     STATE = CheckMembers;
77                     rate.sleep();
78                     break;
79                 }
80
81             case CheckMembers:
82                 {
83                     // check members to elect members transition

```

```

84         ROS_INFO("Mission Control in Check Members state");
85         nh.getParam("/updating_mission", updating_mission);
86         if(updating_mission == 1){
87             update_status_pub.publish(sync); // update status can have 0
88             subs
89             MissionController->possible_member();
90             STATE = ElectMembers;
91             rate.sleep();
92             break;
93         }
94         break;
95     }
96
97     case ElectMembers:
98     {
99         // elect members self loop
100         nh.getParam("/updating_mission", updating_mission);
101         if(updating_mission == 1){
102
103             while(member_election_pub.getNumSubscribers() < 3){
104                 //waiting for sync
105             }
106             ROS_INFO("Setting updating mission false");
107             nh.setParam("/updating_mission", 0);
108             member_election_pub.publish(sync);

```

```

109         STATE = ElectMembers;
110         rate.sleep();
111     }
112
113     nh.getParam("/vote_counter", vote_counter);
114     ROS_INFO("Number of votes in parameter server: %d",
115             vote_counter);
116     while(vote_counter < 3){
117         nh.getParam("/vote_counter", vote_counter);
118     }
119
120     // electmembers to update members transition
121     ROS_INFO("Mission Control in ElectMembers state");
122     nh.getParam("/updating_mission", updating_mission);
123     if(updating_mission == 0){
124         MissionController->elect_members();
125         nh.getParam("/vote_counter", vote_counter);
126         ROS_INFO("After MC member election, vote counter should be
127                 0: %d", vote_counter);
128         STATE = UpdateMembers;
129         rate.sleep();
130         break;
131     }
132     rate.sleep();
133     break;
134 }

```

```

133
134     case UpdateMembers:
135     {
136         // updatemembers to leader election transition
137         ROS_INFO("MC in UpdateMembers, publishing update status. ");
138         update_status_pub.publish(sync); //update status can have 0 subs
139         nh.setParam("/updating_mission", 1);
140         STATE = LeaderElection;
141         rate.sleep();
142         break;
143     }
144
145     case LeaderElection:
146     {
147         // leader election self loop
148         nh.getParam("/vote_counter", vote_counter);
149         nh.getParam("/updating_mission", updating_mission);
150         if(vote_counter == 0 && updating_mission == 1){
151             while(election_pub.getNumSubscribers() < 3){
152                 ROS_INFO("waiting for election subs leader election state");
153             }
154             election_pub.publish(sync);
155             ROS_INFO("Mission Control Sending Leader Election sync");
156             nh.setParam("/updating_mission", 1);
157             rate.sleep();
158         }

```

```

159
160         nh.getParam("/vote_counter", vote_counter);
161         while(vote_counter < 3){
162             // wait till votes are 3
163             nh.getParam("/vote_counter", vote_counter);
164         }
165
166         // leaderelection to missionstarted transition
167         nh.getParam("/Needed", Needed);
168         if(vote_counter == Needed){
169             update_status_pub.publish(sync);
170             MissionController->elect_leader();
171             ROS_INFO("Mission Control Leader elected");
172             STATE = MissionStarted;
173             rate.sleep();
174             break;
175         }
176         rate.sleep();
177         break;
178     }
179
180     case MissionStarted:
181     {
182         // missionstarted to missionaccomplished transition
183         ROS_INFO("MC in mission started state");
184         nh.getParam("/vote_counter", vote_counter);

```

```

185     nh.getParam("/updating_mission", updating_mission);
186     if(MissionController->swarm_reached_goal()){
187         while(mission_end_pub.getNumSubscribers() < 3){
188             ROS_INFO("waiting for mission end subs mission started
189                 state");
190         }
191
192         mission_end_pub.publish(sync);
193         nh.setParam("/updating_mission", 0);
194         STATE = MissionAccomplished;
195         rate.sleep();
196         break;
197     }
198
199     // missionstarted to checkmembers transition
200     location_updated_sub = nh.subscribe("/location_updated", 1,
201         locationUpdatedCallback);
202     while(location_updated_var != 1){
203         ros::spinOnce();
204     }
205     if(location_updated_var == 1){
206         nh.setParam("/updating_mission", 1);
207         nh.setParam("/vote_counter", 0);
208         MissionController->reset_arrays();
209         STATE = CheckMembers;
210         location_updated_sub.shutdown();

```

```

209             location_updated_var = 0;
210             rate.sleep();
211             break;
212         }
213         rate.sleep();
214         break;
215     }
216
217     case MissionAccomplished:
218
219         break;
220
221     }
222     rate.sleep();
223 }
224 ROS_INFO("ROS IS NOT OK");
225 }

```

CodeFiles/MissionControl.cpp